

# User Documentation for ARKODE v6.7.0

SUNDIALS v7.7.0

Daniel R. Reynolds<sup>1</sup>, David J. Gardner<sup>2</sup>, Carol S. Woodward<sup>2</sup>, Cody J. Balos<sup>2</sup> Rujeko Chinomona<sup>3</sup>, and Mustafa

<sup>1</sup>*Department of Mathematics, Southern Methodist University*

<sup>2</sup>*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*

<sup>3</sup>*Department of Mathematics, Temple University*

April 06, 2026



LLNL-SM-668082

### **DISCLAIMER**

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

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

## **CONTRIBUTORS**

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

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



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Changes to SUNDIALS in release 6.7.0 . . . . .	4
1.2	Reading this User Guide . . . . .	7
1.3	SUNDIALS License and Notices . . . . .	7
<b>2</b>	<b>Mathematical Considerations</b>	<b>11</b>
2.1	Adaptive single-step methods . . . . .	12
2.2	Interpolation . . . . .	12
2.3	ARKStep – Additive Runge–Kutta methods . . . . .	14
2.4	ERKStep – Explicit Runge–Kutta methods . . . . .	15
2.5	ForcingStep – Forcing method . . . . .	16
2.6	LSRKStep – Low-Storage Runge–Kutta methods . . . . .	16
2.7	MRISStep – Multirate infinitesimal step methods . . . . .	17
2.8	SplittingStep – Operator splitting methods . . . . .	20
2.9	SPRKStep – Symplectic Partitioned Runge–Kutta methods . . . . .	21
2.10	Error norms . . . . .	22
2.11	Time step adaptivity . . . . .	23
2.12	Initial step size estimation . . . . .	26
2.13	Explicit stability . . . . .	27
2.14	Fixed time stepping . . . . .	28
2.15	Algebraic solvers . . . . .	29
2.16	Rootfinding . . . . .	39
2.17	Inequality Constraints . . . . .	40
2.18	Relaxation Methods . . . . .	40
2.19	Adjoint Sensitivity Analysis . . . . .	41
<b>3</b>	<b>Code Organization</b>	<b>43</b>
<b>4</b>	<b>Getting Started</b>	<b>45</b>
4.1	Data Types . . . . .	46
4.2	The SUNContext Type . . . . .	48
4.3	Error Checking . . . . .	53
4.4	Status and Error Logging . . . . .	56
4.5	Performance Profiling . . . . .	64
4.6	Getting Version Information . . . . .	67
4.7	Features for GPU Accelerated Computing . . . . .	68
<b>5</b>	<b>Using ARKODE</b>	<b>71</b>
5.1	Access to library and header files . . . . .	71
5.2	A skeleton of the user’s main program . . . . .	73
5.3	ARKODE User-callable functions . . . . .	76
5.4	User-supplied functions . . . . .	170
5.5	Relaxation Methods . . . . .	188

5.6	Preconditioner modules . . . . .	193
5.7	Using the ARKStep time-stepping module . . . . .	202
5.8	Using the ERKStep time-stepping module . . . . .	285
5.9	Using the ForcingStep time-stepping module . . . . .	321
5.10	Using the LSRKStep time-stepping module . . . . .	323
5.11	Using the MRISStep time-stepping module . . . . .	333
5.12	Using the SplittingStep time-stepping module . . . . .	392
5.13	Using the SPRKStep time-stepping module . . . . .	403
5.14	Adjoint Sensitivity Analysis . . . . .	418
<b>6</b>	<b>Butcher Table Data Structure</b>	<b>423</b>
6.1	ARKodeButcherTable functions . . . . .	424
<b>7</b>	<b>SPRK Method Table Structure</b>	<b>429</b>
7.1	ARKodeSPRKTable functions . . . . .	430
<b>8</b>	<b>Vector Data Structures</b>	<b>433</b>
8.1	Description of the NVECTOR Modules . . . . .	433
8.2	Description of the NVECTOR operations . . . . .	441
8.3	NVECTOR functions required by ARKODE . . . . .	453
8.4	The NVECTOR_SERIAL Module . . . . .	455
8.5	The NVECTOR_PARALLEL Module . . . . .	458
8.6	The NVECTOR_OPENMP Module . . . . .	461
8.7	The NVECTOR_PTHREADS Module . . . . .	465
8.8	The NVECTOR_PARHYP Module . . . . .	468
8.9	The NVECTOR_PETSC Module . . . . .	470
8.10	The NVECTOR_CUDA Module . . . . .	473
8.11	The NVECTOR_HIP Module . . . . .	478
8.12	The NVECTOR_SYCL Module . . . . .	483
8.13	The NVECTOR_RAJA Module . . . . .	488
8.14	The NVECTOR_KOKKOS Module . . . . .	491
8.15	The NVECTOR_OPENMPDEV Module . . . . .	494
8.16	The NVECTOR_TRILINOS Module . . . . .	497
8.17	The NVECTOR_MANYVECTOR Module . . . . .	498
8.18	The NVECTOR_MPIMANYVECTOR Module . . . . .	501
8.19	The NVECTOR_MPIPLUSX Module . . . . .	504
8.20	NVECTOR Examples . . . . .	506
<b>9</b>	<b>Matrix Data Structures</b>	<b>511</b>
9.1	Description of the SUNMATRIX Modules . . . . .	511
9.2	Description of the SUNMATRIX operations . . . . .	514
9.3	The SUNMATRIX_DENSE Module . . . . .	516
9.4	The SUNMATRIX_MAGMADENSE Module . . . . .	519
9.5	The SUNMATRIX_ONEMKLDENSE Module . . . . .	523
9.6	The SUNMATRIX_BAND Module . . . . .	528
9.7	The SUNMATRIX_CUSPARSE Module . . . . .	534
9.8	The SUNMATRIX_SPARSE Module . . . . .	537
9.9	The SUNMATRIX_SLUNRLOC Module . . . . .	543
9.10	The SUNMATRIX_GINKGO Module . . . . .	545
9.11	The SUNMATRIX_GINKGOBATCH Module . . . . .	547
9.12	The SUNMATRIX_KOKKOSDENSE Module . . . . .	549
9.13	SUNMATRIX Examples . . . . .	552
9.14	SUNMATRIX functions used by ARKODE . . . . .	553
<b>10</b>	<b>Linear Algebraic Solvers</b>	<b>555</b>

10.1	The SUNLinearSolver API . . . . .	556
10.2	ARKODE SUNLinearSolver interface . . . . .	569
10.3	The SUNLinSol_Band Module . . . . .	572
10.4	The SUNLinSol_Dense Module . . . . .	573
10.5	The SUNLinSol_KLU Module . . . . .	575
10.6	The SUNLinSol_LapackBand Module . . . . .	578
10.7	The SUNLinSol_LapackDense Module . . . . .	580
10.8	The SUNLinSol_MagmaDense Module . . . . .	582
10.9	The SUNLinSol_OneMklDense Module . . . . .	583
10.10	The SUNLinSol_PCG Module . . . . .	584
10.11	The SUNLinSol_SPBCGS Module . . . . .	588
10.12	The SUNLinSol_SPGMR Module . . . . .	591
10.13	The SUNLinSol_SPGMR Module . . . . .	595
10.14	The SUNLinSol_SPTFQMR Module . . . . .	599
10.15	The SUNLinSol_SuperLUDIST Module . . . . .	603
10.16	The SUNLinSol_SuperLUMT Module . . . . .	606
10.17	The SUNLinSol_cuSolverSp_batchQR Module . . . . .	608
10.18	The SUNLINEARSOLVER_GINKGO Module . . . . .	610
10.19	The SUNLINEARSOLVER_GINKGOBATCH Module . . . . .	613
10.20	The SUNLINEARSOLVER_KOKKOSDENSE Module . . . . .	618
10.21	SUNLinearSolver Examples . . . . .	620
<b>11</b>	<b>Nonlinear Algebraic Solvers</b>	<b>623</b>
11.1	The SUNNonlinearSolver API . . . . .	623
11.2	ARKODE SUNNonlinearSolver interface . . . . .	633
11.3	The SUNNonlinSol_Newton implementation . . . . .	639
11.4	The SUNNonlinSol_FixedPoint implementation . . . . .	641
11.5	The SUNNonlinSol_PetscSNES implementation . . . . .	645
<b>12</b>	<b>Dominant Eigenvalue Estimators</b>	<b>649</b>
12.1	Introduction to Dominant Eigenvalue Estimators . . . . .	649
12.2	The SUNDomEigEstimator API . . . . .	649
12.3	SUNDIALS modules SUNDomEigEstimator interface . . . . .	656
12.4	The SUNDomEigEstimator_Power Module . . . . .	657
12.5	The SUNDomEigEstimator_Arnoldi Module . . . . .	660
<b>13</b>	<b>Time Step Adaptivity Controllers</b>	<b>663</b>
13.1	The SUNAdaptController API . . . . .	663
13.2	The SUNAdaptController_Soderlind Module . . . . .	670
13.3	The SUNAdaptController_ImExGus Module . . . . .	677
13.4	The SUNAdaptController_MRIHTol Module . . . . .	679
<b>14</b>	<b>Stepper Data Structure</b>	<b>683</b>
14.1	The SUNStepper API . . . . .	683
14.2	Implementing a SUNStepper . . . . .	691
<b>15</b>	<b>Adjoint Sensitivity Analysis</b>	<b>693</b>
15.1	Introduction to Adjoint Sensitivity Analysis . . . . .	693
15.2	The SUNAdjointStepper Class . . . . .	694
15.3	The SUNAdjointCheckpointScheme Class . . . . .	698
15.4	The SUNAdjointCheckpointScheme_Fixed Module . . . . .	702
<b>16</b>	<b>Tools for Memory Management</b>	<b>705</b>
16.1	The SUNMemoryHelper API . . . . .	705
16.2	The SUNMemoryHelper_Sys Implementation . . . . .	711

16.3	The SUNMemoryHelper_Cuda Implementation . . . . .	712
16.4	The SUNMemoryHelper_Hip Implementation . . . . .	714
16.5	The SUNMemoryHelper_Sycl Implementation . . . . .	717
<b>17</b>	<b>Installing SUNDIALS</b>	<b>721</b>
17.1	Installing with Spack . . . . .	721
17.2	Installing with CMake . . . . .	721
17.3	Configuration options . . . . .	724
17.4	Testing the Build and Installation . . . . .	753
17.5	Building and Running Examples . . . . .	753
17.6	Using SUNDIALS In Your Project . . . . .	754
17.7	Libraries and Header Files . . . . .	755
<b>18</b>	<b>ARKODE Constants</b>	<b>775</b>
<b>19</b>	<b>Butcher Tables</b>	<b>781</b>
19.1	Explicit Butcher tables . . . . .	782
19.2	Implicit Butcher tables . . . . .	803
19.3	Additive Butcher tables . . . . .	823
19.4	Symplectic Partitioned Butcher tables . . . . .	823
<b>20</b>	<b>Fortran</b>	<b>827</b>
20.1	Introduction . . . . .	827
20.2	Data Types . . . . .	829
20.3	Notable Fortran/C usage differences . . . . .	830
20.4	Common Issues . . . . .	835
<b>21</b>	<b>Python</b>	<b>837</b>
21.1	Using sundials4py . . . . .	837
21.2	core Submodule . . . . .	848
21.3	arkode Submodule . . . . .	878
<b>22</b>	<b>Release History</b>	<b>905</b>
<b>23</b>	<b>Changelog</b>	<b>907</b>
23.1	Changes to SUNDIALS in release 7.7.0 . . . . .	907
23.2	Changes to SUNDIALS in release 7.6.0 . . . . .	910
23.3	Changes to SUNDIALS in release 7.5.0 . . . . .	911
23.4	Changes to SUNDIALS in release 7.4.0 . . . . .	912
23.5	Changes to SUNDIALS in release 7.3.0 . . . . .	912
23.6	Changes to SUNDIALS in release 7.2.1 . . . . .	915
23.7	Changes to SUNDIALS in release 7.2.0 . . . . .	915
23.8	Changes to SUNDIALS in release 7.1.1 . . . . .	918
23.9	Changes to SUNDIALS in release 7.1.0 . . . . .	918
23.10	Changes to SUNDIALS in release 7.0.0 . . . . .	920
23.11	Changes to SUNDIALS in release 6.7.0 . . . . .	923
23.12	Changes to SUNDIALS in release 6.6.2 . . . . .	924
23.13	Changes to SUNDIALS in release 6.6.1 . . . . .	924
23.14	Changes to SUNDIALS in release 6.6.0 . . . . .	924
23.15	Changes to SUNDIALS in release 6.5.1 . . . . .	925
23.16	Changes to SUNDIALS in release 6.5.0 . . . . .	925
23.17	Changes to SUNDIALS in release 6.4.1 . . . . .	926
23.18	Changes to SUNDIALS in release 6.4.0 . . . . .	926
23.19	Changes to SUNDIALS in release 6.3.0 . . . . .	927
23.20	Changes to SUNDIALS in release 6.2.0 . . . . .	927



23.21 Changes to SUNDIALS in release 6.1.1 . . . . .	930
23.22 Changes to SUNDIALS in release 6.1.0 . . . . .	931
23.23 Changes to SUNDIALS in release 6.0.0 . . . . .	931
23.24 Changes to SUNDIALS in release 5.8.0 . . . . .	937
23.25 Changes to SUNDIALS in release 5.7.0 . . . . .	938
23.26 Changes to SUNDIALS in release 5.6.1 . . . . .	938
23.27 Changes to SUNDIALS in release 5.6.0 . . . . .	938
23.28 Changes to SUNDIALS in release 5.5.0 . . . . .	938
23.29 Changes to SUNDIALS in release 5.4.0 . . . . .	939
23.30 Changes to SUNDIALS in release 5.3.0 . . . . .	941
23.31 Changes to SUNDIALS in release 5.2.0 . . . . .	942
23.32 Changes to SUNDIALS in release 5.1.0 . . . . .	943
23.33 Changes to SUNDIALS in release 5.0.0 . . . . .	943
23.34 Changes to SUNDIALS in release 4.1.0 . . . . .	947
23.35 Changes to SUNDIALS in release 4.0.2 . . . . .	947
23.36 Changes to SUNDIALS in release 4.0.1 . . . . .	948
23.37 Changes to SUNDIALS in release 4.0.0 . . . . .	948
23.38 Changes to SUNDIALS in release 3.2.1 . . . . .	950
23.39 Changes to SUNDIALS in release 3.2.0 . . . . .	951
23.40 Changes to SUNDIALS in release 3.1.2 . . . . .	951
23.41 Changes to SUNDIALS in release 3.1.1 . . . . .	952
23.42 Changes to SUNDIALS in release 3.1.0 . . . . .	953
23.43 Changes to SUNDIALS in release 3.0.0 . . . . .	953
23.44 Changes to SUNDIALS in release 2.7.0 . . . . .	955
23.45 Changes to SUNDIALS in release 2.6.2 . . . . .	957
23.46 Changes to SUNDIALS in release 2.6.1 . . . . .	957
23.47 Changes to SUNDIALS in release 2.6.0 . . . . .	957
23.48 Changes to SUNDIALS in release 2.5.0 . . . . .	959
23.49 Changes to SUNDIALS in release 2.4.0 . . . . .	960
23.50 Changes to SUNDIALS in release 2.3.0 . . . . .	960
23.51 Changes to SUNDIALS in release 2.2.0 . . . . .	961
23.52 Changes to SUNDIALS in release 2.1.1 . . . . .	962
23.53 Changes to SUNDIALS in release 2.1.0 . . . . .	962
23.54 Changes to SUNDIALS in release 2.0.2 . . . . .	962
23.55 Changes to SUNDIALS in release 2.0.1 . . . . .	962
23.56 Changes to SUNDIALS in release 2.0.0 . . . . .	962
<b>Bibliography</b>	<b>965</b>
<b>Python Module Index</b>	<b>973</b>
<b>Index</b>	<b>975</b>



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

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

ARKODE is developed by [Southern Methodist University](#) and [Lawrence Livermore National Security](#), with support by the [US Department of Energy, Office of Science](#).



# Chapter 1

## Introduction

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

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

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

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

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

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

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

This framework allows for significant freedom over the constitutive methods used for each component, and ARKODE is packaged with a wide array of built-in methods for use. These built-in Butcher tables include adaptive explicit methods of orders 2-9, adaptive implicit methods of orders 2-5, and adaptive ImEx methods of orders 2-5.

*ERKStep* focuses specifically on problems posed in explicit form,

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

allowing for increased computational efficiency and memory savings. The algorithms used in *ERKStep* are adaptive- and fixed-step explicit Runge–Kutta methods. As with *ARKStep*, the *ERKStep* module is packaged with adaptive explicit methods of orders 2-9.

*SPRKStep* focuses on Hamiltonian systems posed in the form,

$$H(t, p, q) = T(t, p) + V(t, q)$$

$$\dot{p} = f_1(t, q) = \frac{\partial V(t, q)}{\partial q}, \quad \dot{q} = f_2(t, p) = \frac{\partial T(t, p)}{\partial p}, \quad (1.3)$$

allowing for conservation of quadratic invariants.

*MRIS*Step focuses specifically on problems posed in additive form,

$$\dot{y} = f^E(t, y) + f^I(t, y) + f^F(t, y), \quad y(t_0) = y_0. \quad (1.4)$$

where here the right-hand side function is additively split into three components:

- $f^E(t, y)$  contains the “slow-nonstiff” components of the system (this will be integrated using an explicit method and a large time step  $h^S$ ),
- $f^I(t, y)$  contains the “slow-stiff” components of the system (this will be integrated using an implicit method and a large time step  $h^S$ ), and
- $f^F(t, y)$  contains the “fast” components of the system (this will be integrated using a possibly different method than the slow time scale and a small time step  $h^F \ll h^S$ ).

For such problems, *MRIS*Step provides fixed-step slow step multirate infinitesimal step (MIS), multirate infinitesimal GARK (MRI-GARK), and implicit-explicit MRI-GARK (IMEX-MRI-GARK) methods, allowing for evolution of the problem (1.4) using multirate methods having orders of accuracy 2-4.

For ARKStep or *MRIS*Step 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 *SUNNonlinearSolver* module, supplied either by the user or from the underlying *SUNDIALS* infrastructure. For nonlinear solver algorithms that internally require a linear solver, ARKODE may use a variety of *SUNLinearSolver* modules provided with *SUNDIALS*, or again may utilize a user-supplied module.

## 1.1 Changes to SUNDIALS in release 6.7.0

### New Features and Enhancements

The default number of stages for the SSP Runge-Kutta methods [ARKODE\\_LSRK\\_SSP\\_S\\_2](#) and [ARKODE\\_LSRK\\_SSP\\_S\\_3](#) in *LSRK*Step were changed from 10 and 9, respectively, to their minimum allowable values of 2 and 4. Users may revert to the previous values by calling [LSRKStepSetNumSSPStages\(\)](#).

Added the optional function [ARKodeInit\(\)](#) to ARKODE to enable data allocation before the first call to [ARKodeEvolve\(\)](#) (but after all other optional input routines have been called), to support users who measure memory usage before beginning a simulation.

Added the function [ARKodeGetStageIndex\(\)](#) that returns the index of the stage currently being processed, and the total number of stages in the method, for users who wish to compute auxiliary quantities in their IVP right-hand side functions during some stages and not others (e.g., in all but the first or last stage).

Added the functions [ARKodeGetLastTime\(\)](#) and [ARKodeGetLastState\(\)](#) to return the last successful time and state achieved by ARKODE, respectively.

ARKODE now allows users to supply functions that will be called before each internal time step attempt ([ARKodeSetPreStepFn\(\)](#)), after each successful time step ([ARKodeSetPostStepFn\(\)](#)), before right-hand side routines are called on an updated state ([ARKodeSetPreRhsFn\(\)](#)), and/or once each internal step/stage is computed ([ARKodeSetPostprocessStepFn\(\)](#)/[ARKodeSetPostprocessStageFn\(\)](#)). These are considered **advanced** functions, as they should treat the state vector as read-only, otherwise all theoretical guarantees of solution accuracy and stability will be lost. As a result of these new functions, the values of multiple ARKODE return codes (e.g., `ARK_INTERP_FAIL`) have been updated; users who key off of the named constants will not be affected, but users who rely on the values themselves should update their codes accordingly.

Note to users utilizing the previously undocumented `ARKodeSetPostprocessStepFn()` function, the supplied function is now called on the newly computed state vector for all step attempts not just successful steps. To obtain the previous behavior of only calling a function on successful steps, switch to using `ARKodeSetPostStepFn()`.

Added `SUNLogger_Set{Error,Warning,Info,Debug}File` functions to allow setting logger output streams with a FILE\*.

Updated the Kokkos N\_Vector to support Kokkos 5.x versions.

### Bug Fixes

Fixed a CMake bug where the SuperLU\_MT interface would not be built and installed without setting the `SUPERLUMT_WORKS` option to `TRUE`.

Fixed the embedded coefficients for the `ARKODE_TSITOURAS_7_4_5` Butcher table.

Fixed a bug in `LSRKStep` where an incorrect state vector could be passed to a user-supplied dominant eigenvalue function on the first step unless the output vector passed to `ARKodeEvolve()` contained the initial condition and when an eigenvalue estimate is requested on the first step in a subsequent call to `ARKodeEvolve()` unless the output vector passed contained the most recently returned solution.

Fixed a potential bug in `LSRKStep`'s `ARKODE_LSRK_SSP_S_3` method, where a real number was used instead of an integer, potentially resulting in a rounding error.

Fixed a bug in `MRISStep` for estimating the first “slow” time step in an adaptive multirate calculation.

Fixed a bug in `MRISStep` when using a custom inner integrator that relies on the input state being the initial condition for the fast integration rather than retaining the result from the last inner integration or most recent reset call and the output vector passed to `ARKodeEvolve()` does not contain the initial condition on the first call or the last returned solution on subsequent calls.

Added a missing call to `SUNNonlinSolSetup()` in `MRISStep` when using an IMEX-MRI-SR method.

Fixed a bug in the ARKODE discrete adjoint checkpointing where an incorrect state would be stored on the first step if the output vector passed to `ARKodeEvolve()` did not contain the initial condition on the first call.

Removed extraneous copy of output vector when using ARKODE in `ARK_ONE_STEP` mode.

Removed an extraneous copy of the output vector in each step with `SplittingStep`.

Fixed a bug in logging output from ARKODE, where for some time stepping modules, the current “time” output in the logger was incorrect.

Fixed a bug where passing an empty string to `SUNLogger_Set{Error,Warning,Info,Debug}Filename` did not disable the corresponding logging stream [Issue #844](#).

### Deprecation Notices

The `CVodeSetMonitorFn` and `CVodeSetMonitorFrequency` functions have been deprecated and will be removed in the next major release.

Several CMake options have been deprecated in favor of namespaced versions prefixed with `SUNDIALS_` to avoid naming collisions in applications that include SUNDIALS directly within their CMake builds. Additionally, a consistent naming convention (`SUNDIALS_ENABLE`) is now used for all boolean options. The table below lists the old CMake option names and the new replacements.

Old Option	New Option
<code>BUILD_ARKODE</code>	<code>SUNDIALS_ENABLE_ARKODE</code>
<code>BUILD_CVODE</code>	<code>SUNDIALS_ENABLE_CVODE</code>
<code>BUILD_CVODES</code>	<code>SUNDIALS_ENABLE_CVODES</code>
<code>BUILD_IDA</code>	<code>SUNDIALS_ENABLE_IDA</code>

continues on next page

Table 1.1 – continued from previous page

BUILD_IDAS	<i>SUNDIALS_ENABLE_IDAS</i>
BUILD_KINSOL	<i>SUNDIALS_ENABLE_KINSOL</i>
ENABLE_MPI	<i>SUNDIALS_ENABLE_MPI</i>
ENABLE_OPENMP	<i>SUNDIALS_ENABLE_OPENMP</i>
ENABLE_OPENMP_DEVICE	<i>SUNDIALS_ENABLE_OPENMP_DEVICE</i>
OPENMP_DEVICE_WORKS	<i>SUNDIALS_ENABLE_OPENMP_DEVICE_CHECKS</i>
ENABLE_PTHREAD	<i>SUNDIALS_ENABLE_PTHREAD</i>
ENABLE_CUDA	<i>SUNDIALS_ENABLE_CUDA</i>
ENABLE_HIP	<i>SUNDIALS_ENABLE_HIP</i>
ENABLE_SYCL	<i>SUNDIALS_ENABLE_SYCL</i>
ENABLE_LAPACK	<i>SUNDIALS_ENABLE_LAPACK</i>
LAPACK_WORKS	<i>SUNDIALS_ENABLE_LAPACK_CHECKS</i>
ENABLE_GINKGO	<i>SUNDIALS_ENABLE_GINKGO</i>
GINKGO_WORKS	<i>SUNDIALS_ENABLE_GINKGO_CHECKS</i>
ENABLE_MAGMA	<i>SUNDIALS_ENABLE_MAGMA</i>
MAGMA_WORKS	<i>SUNDIALS_ENABLE_MAGMA_CHECKS</i>
ENABLE_SUPERLUDIST	<i>SUNDIALS_ENABLE_SUPERLUDIST</i>
SUPERLUDIST_WORKS	<i>SUNDIALS_ENABLE_SUPERLUDIST_CHECKS</i>
ENABLE_SUPERLUMT	<i>SUNDIALS_ENABLE_SUPERLUMT</i>
SUPERLUMT_WORKS	<i>SUNDIALS_ENABLE_SUPERLUMT_CHECKS</i>
ENABLE_KLU	<i>SUNDIALS_ENABLE_KLU</i>
KLU_WORKS	<i>SUNDIALS_ENABLE_KLU_CHECKS</i>
ENABLE_HYPRE	<i>SUNDIALS_ENABLE_HYPRE</i>
HYPRE_WORKS	<i>SUNDIALS_ENABLE_HYPRE_CHECKS</i>
ENABLE_PETSC	<i>SUNDIALS_ENABLE_PETSC</i>
PETSC_WORKS	<i>SUNDIALS_ENABLE_PETSC_CHECKS</i>
ENABLE_TRILINOS	<i>SUNDIALS_ENABLE_TRILINOS</i>
ENABLE_RAJA	<i>SUNDIALS_ENABLE_RAJA</i>
ENABLE_XBRAID	<i>SUNDIALS_ENABLE_XBRAID</i>
XBRAID_WORKS	<i>SUNDIALS_ENABLE_XBRAID_CHECKS</i>
ENABLE_ONEMKL	<i>SUNDIALS_ENABLE_ONEMKL</i>
ONEMKL_WORKS	<i>SUNDIALS_ENABLE_ONEMKL_CHECKS</i>
ENABLE_CALIPER	<i>SUNDIALS_ENABLE_CALIPER</i>
ENABLE_ADIK	<i>SUNDIALS_ENABLE_ADIK</i>
ENABLE_KOKKOS	<i>SUNDIALS_ENABLE_KOKKOS</i>
KOKKOS_WORKS	<i>SUNDIALS_ENABLE_KOKKOS_CHECKS</i>
ENABLE_KOKKOS_KERNELS	<i>SUNDIALS_ENABLE_KOKKOS_KERNELS</i>
KOKKOS_KERNELS_WORKS	<i>SUNDIALS_ENABLE_KOKKOS_KERNELS_CHECKS</i>
BUILD_FORTRAN_MODULE_INTERFACE	<i>SUNDIALS_ENABLE_FORTRAN</i>
SUNDIALS_BUILD_WITH_PROFILING	<i>SUNDIALS_ENABLE_PROFILING</i>
SUNDIALS_BUILD_WITH_MONITORING	<i>SUNDIALS_ENABLE_MONITORING</i>
SUNDIALS_BUILD_PACKAGE_FUSED_KERNELS	<i>SUNDIALS_ENABLE_PACKAGE_FUSED_KERNELS</i>
EXAMPLES_ENABLE_C	<i>SUNDIALS_ENABLE_C_EXAMPLES</i>
EXAMPLES_ENABLE_CXX	<i>SUNDIALS_ENABLE_CXX_EXAMPLES</i>
EXAMPLES_ENABLE_F2003	<i>SUNDIALS_ENABLE_FORTRAN_EXAMPLES</i>
EXAMPLES_ENABLE_CUDA	<i>SUNDIALS_ENABLE_CUDA_EXAMPLES</i>
EXAMPLES_INSTALL	<i>SUNDIALS_ENABLE_EXAMPLES_INSTALL</i>
EXAMPLES_INSTALL_PATH	<i>SUNDIALS_EXAMPLES_INSTALL_PATH</i>
BUILD_BENCHMARKS	<i>SUNDIALS_ENABLE_BENCHMARKS</i>
BENCHMARKS_INSTALL_PATH	<i>SUNDIALS_BENCHMARKS_INSTALL_PATH</i>
SUNDIALS_BENCHMARK_OUTPUT_DIR	<i>SUNDIALS_BENCHMARKS_OUTPUT_DIR</i>
SUNDIALS_BENCHMARK_CALIPER_OUTPUT_DIR	<i>SUNDIALS_BENCHMARKS_CALIPER_OUTPUT_DIR</i>

continues on next page



Table 1.1 – continued from previous page

SUNDIALS_BENCHMARK_NUM_CPUS	SUNDIALS_BENCHMARKS_NUM_CPUS
SUNDIALS_BENCHMARK_NUM_GPUS	SUNDIALS_BENCHMARKS_NUM_GPUS
ENABLE_ALL_WARNINGS	SUNDIALS_ENABLE_ALL_WARNINGS
ENABLE_WARNINGS_AS_ERRORS	CMAKE_COMPILE_WARNING_AS_ERROR
ENABLE_ADDRESS_SANITIZER	SUNDIALS_ENABLE_ADDRESS_SANITIZER
ENABLE_MEMORY_SANITIZER	SUNDIALS_ENABLE_MEMORY_SANITIZER
ENABLE_LEAK_SANITIZER	SUNDIALS_ENABLE_LEAK_SANITIZER

Following the updated CMake options, the macros listed below have been deprecated and replaced with versions that align with the new CMake options.

Old Macro	New Macro
SUNDIALS_BUILD_WITH_PROFILING	SUNDIALS_ENABLE_PROFILING
SUNDIALS_BUILD_WITH_MONITORING	SUNDIALS_ENABLE_MONITORING
SUNDIALS_BUILD_PACKAGE_FUSED_KERNELS	SUNDIALS_ENABLE_PACKAGE_FUSED_KERNELS

For changes in prior versions of SUNDIALS see §23.

## 1.2 Reading this User Guide

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

The structure of this document is as follows:

- In the next section we provide a thorough presentation of the underlying *mathematical algorithms* used within the ARKODE family of solvers.
- We follow this with an overview of how the source code for ARKODE is *organized*.
- The largest section follows, providing a full account of how to use ARKODE within C and C++ applications, including any instructions that are specific to a given time-stepping modules, *ARKStep*, *ERKStep*, or *MRIStep*. This section then includes additional information on how to use ARKODE from applications written in *Fortran*, as well as information on how to leverage *GPU accelerators within ARKODE*.
- A much smaller section follows, describing ARKODE’s *Butcher table structure*, that is used by both ARKStep and ERKStep.
- Subsequent sections discuss shared SUNDIALS features that are used by ARKODE: *vector data structures*, *matrix data structures*, *linear solver data structures*, *nonlinear solver data structures*, *memory management utilities*, 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 License and Notices

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

**Note**

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

### 1.3.1 BSD 3-Clause License

Copyright (c) 2025, Lawrence Livermore National Security, University of Maryland Baltimore County, and the SUNDIALS contributors. Copyright (c) 2013-2025, Lawrence Livermore National Security and Southern Methodist University. Copyright (c) 2002-2013, Lawrence Livermore National Security. All rights reserved.

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder 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 THE COPYRIGHT HOLDER 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.

### 1.3.2 Additional Notice

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

This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC.

The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

### **1.3.3 SUNDIALS Release Numbers**

LLNL-CODE-667205 (ARKODE)

UCRL-CODE-155951 (CVODE)

UCRL-CODE-155950 (CVODES)

UCRL-CODE-155952 (IDA)

UCRL-CODE-237203 (IDAS)

LLNL-CODE-665877 (KINSOL)



## Chapter 2

# Mathematical Considerations

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

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

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

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

The ODE right-hand side is given by the function  $f(t, y)$  – in general we make no assumption that the problem (2.1) is autonomous (i.e.,  $f = f(y)$ ) or linear ( $f = Ay$ ). In general, the time integration methods within ARKODE support additive splittings of this right-hand side function, as described in the subsections that follow. Through these splittings, the time-stepping methods currently supplied with ARKODE are designed to solve stiff, nonstiff, mixed stiff/nonstiff, and multirate problems. As per Ascher and Petzold [13], a problem is “stiff” if the stepsize needed to maintain stability of the forward Euler method is much smaller than that required to represent the solution accurately.

In the sub-sections that follow, we elaborate on the numerical methods utilized in ARKODE. We first discuss the “single-step” nature of the ARKODE infrastructure, including its usage modes and approaches for interpolated solution output. We then discuss the current suite of time-stepping modules supplied with ARKODE, including

- ARKStep for *additive Runge–Kutta methods*
- ERKStep that is optimized for *explicit Runge–Kutta methods*
- ForcingStep for *a forcing method*
- LSRKStep that supports *low-storage Runge–Kutta methods*
- MRISStep for *multirate infinitesimal step (MIS), multirate infinitesimal GARK (MRI-GARK), and implicit-explicit MRI-GARK (IMEX-MRI-GARK) methods*
- SplittingStep for *operator splitting methods*
- SPRKStep for *symplectic partitioned Runge–Kutta methods*

We then discuss the *adaptive temporal error controllers* shared by the time-stepping modules, including discussion of our choice of norms for measuring errors within various components of the solver.

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

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

## 2.1 Adaptive single-step methods

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

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

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

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

ARKODE may be run in a variety of “modes”:

- **NORMAL** – The solver will take internal steps until it has just overtaken a user-specified output time,  $t_{\text{out}}$ , in the direction of integration, i.e.  $t_{n-1} < t_{\text{out}} \leq t_n$  for forward integration, or  $t_n \leq t_{\text{out}} < t_{n-1}$  for backward integration. It will then compute an approximation to the solution  $y(t_{\text{out}})$  by interpolation (using one of the dense output routines described in the section §2.2).
- **ONE-STEP** – The solver will only take a single internal step  $y_{n-1} \rightarrow y_n$  and then return control back to the calling program. If this step will overtake  $t_{\text{out}}$  then the solver will again return an interpolated result; otherwise it will return a copy of the internal solution  $y_n$ .
- **NORMAL-TSTOP** – The solver will take internal steps until the next step will overtake  $t_{\text{out}}$ . It will then limit this next step so that  $t_n = t_{n-1} + h_n = t_{\text{out}}$ , and once the step completes it will return a copy of the internal solution  $y_n$ .
- **ONE-STEP-TSTOP** – The solver will check whether the next step will overtake  $t_{\text{out}}$  – if not then this mode is identical to “one-step” above; otherwise it will limit this next step so that  $t_n = t_{n-1} + h_n = t_{\text{out}}$ . In either case, once the step completes it will return a copy of the internal solution  $y_n$ .

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

## 2.2 Interpolation

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

ARKODE provides two complementary interpolation approaches: “Hermite” and “Lagrange”. The former approach has been included with ARKODE since its inception, and is more suitable for non-stiff problems; the latter is a more recent approach that is designed to provide increased accuracy when integrating stiff problems. Both are described in detail below.

### 2.2.1 Hermite interpolation module

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

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

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

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

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

- $q = 0$ : constant interpolant

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

- $q = 1$ : linear Lagrange interpolant

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

- $q = 2$ : quadratic Hermite interpolant

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

- $q = 3$ : cubic Hermite interpolant

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

- $q = 4$ : quartic Hermite interpolant

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

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

- $q = 5$ : quintic Hermite interpolant

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

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

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

## 2.2.2 Lagrange interpolation module

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

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

$$p_j(t) = \prod_{\substack{l=0 \\ l \neq j}}^{\nu} \left( \frac{t - t_l}{t_j - t_l} \right), \quad j = 0, \dots, \nu.$$

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

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

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

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

## 2.3 ARKStep – Additive Runge–Kutta methods

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

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

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

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

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

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

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

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

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

$$y_n = y_{n-1} + h_n \sum_{i=1}^s \left( b_i^E \hat{f}^E(t_{n,i}^E, z_i) + b_i^I \hat{f}^I(t_{n,i}^I, z_i) \right), \quad (2.4)$$

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



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

The user of ARKStep must choose appropriately between one of three classes of methods: *ImEx*, *explicit*, and *implicit*. All of the built-in Butcher tables encoding the coefficients  $c^E$ ,  $c^I$ ,  $A^E$ ,  $A^I$ ,  $b^E$ ,  $b^I$ ,  $\tilde{b}^E$  and  $\tilde{b}^I$  are further described in the section §19.

For mixed stiff/nonstiff problems, a user should provide both of the functions  $f^E$  and  $f^I$  that define the IVP system. For such problems, ARKStep currently implements the ARK methods proposed in [50, 68, 71], allowing for methods having order of accuracy  $q = \{2, 3, 4, 5\}$  and embeddings with orders  $p = \{1, 2, 3, 4\}$ ; the tables for these methods are given in section §19.3. Additionally, user-defined ARK tables are supported.

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

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

In this scenario, the coefficients  $A^I = 0$ ,  $c^I = 0$ ,  $b^I = 0$  and  $\tilde{b}^I = 0$  in (2.4), and the ARK methods reduce to classical explicit Runge–Kutta methods (ERK). For these classes of methods, ARKODE provides coefficients with orders of accuracy  $q = \{2, 3, 4, 5, 6, 7, 8, 9\}$ , with embeddings of orders  $p = \{1, 2, 3, 4, 5, 6, 7, 8\}$ ; the tables for these methods are given in section §19.1. As with ARK methods, user-defined ERK tables are supported.

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

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

Similarly to ERK methods, in this scenario the coefficients  $A^E = 0$ ,  $c^E = 0$ ,  $b^E = 0$  and  $\tilde{b}^E = 0$  in (2.4), and the ARK methods reduce to classical diagonally-implicit Runge–Kutta methods (DIRK). For these classes of methods, ARKODE provides tables with orders of accuracy  $q = \{2, 3, 4, 5\}$ , with embeddings of orders  $p = \{1, 2, 3, 4\}$ ; the tables for these methods are given in section §19.2. Again, user-defined DIRK tables are supported.

## 2.4 ERKStep – Explicit Runge–Kutta methods

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

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

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

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

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

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

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

## 2.5 ForcingStep – Forcing method

The ForcingStep time-stepping module in ARKODE is designed for IVPs of the form

$$\dot{y} = f_1(t, y) + f_2(t, y), \quad y(t_0) = y_0,$$

with two additive partitions. One step of the forcing method implemented in ForcingStep is given by

$$\begin{aligned} v_1(t_{n-1}) &= y_{n-1}, \\ \dot{v}_1 &= f_1(t, v_1), \\ f_1^* &= \frac{v_1(t_n) - y_{n-1}}{h_n}, \\ v_2(t_{n-1}) &= y_{n-1}, \\ \dot{v}_2 &= f_1^* + f_2(t, v_2), \\ y_n &= v_2(t_n). \end{aligned}$$

Like a Lie–Trotter method from *SplittingStep*, the partitions are evolved through a sequence of inner IVPs which can be solved with an arbitrary integrator or exact solution procedure. However, the IVP for partition two includes a “forcing” or “tendency” term  $f_1^*$  to strengthen the coupling. This coupling leads to a first order method provided  $v_1$  and  $v_2$  are integrated to at least first order accuracy. Currently, a fixed time step must be specified for the overall ForcingStep integrator, but partition integrators are free to use adaptive time steps.

## 2.6 LSRKStep – Low-Storage Runge–Kutta methods

The LSRKStep time-stepping module in ARKODE supports a variety of so-called “low-storage” Runge–Kutta (LSRK) methods, [44, 73, 82, 122]. This category includes traditional explicit fixed-step and low-storage Runge–Kutta methods, adaptive low-storage Runge–Kutta methods, and others. These are characterized by coefficient tables that have an exploitable structure, such that their implementation does not require that all stages be stored simultaneously. At present, this module supports explicit, adaptive “super-time-stepping” (STS) and “strong-stability-preserving” (SSP) methods.

The LSRK time-stepping module in ARKODE currently supports IVP of the form (2.7), i.e., unlike the more general problem form (2.2), LSRKStep requires that problems have an identity mass matrix (i.e.,  $M(t) = I$ ) and that the right-hand side function is not split into separate components.

LSRKStep currently supports two families of second-order, explicit, and temporally adaptive STS methods: Runge–Kutta–Chebyshev (RKC), [122] and Runge–Kutta–Legendre (RKL), [82]. These methods have the form

$$\begin{aligned} z_1 &= y_n, \\ z_2 &= z_1 + h_n \tilde{\mu}_2 f(t_n, z_1), \\ z_j &= (1 - \mu_j - \nu_j) z_1 + \mu_j z_{j-1} + \nu_j z_{j-2} + h_n \tilde{\gamma}_j f(t_n, z_1) + h_n \tilde{\mu}_j f(t_n + c_{j-1} h, z_{j-1}), \quad j = 3, \dots, s+1 \\ y_{n+1} &= z_{s+1}. \end{aligned} \tag{2.9}$$

The corresponding coefficients can be found in [122] and [82], respectively.

LSRK methods of STS type are designed for stiff problems characterized by having Jacobians with eigenvalues that have large real and small imaginary parts. While those problems are traditionally treated using implicit methods, STS methods are explicit. To achieve stability for these stiff problems, STS methods use more stages than conventional Runge-Kutta methods to extend the stability region along the negative real axis. The extent of this stability region is proportional to the square of the number of stages used.

LSRK methods of the SSP type are designed to preserve the so-called “strong-stability” properties of advection-type equations. For details, see [73]. The SSPRK methods in ARKODE use the following Shu–Osher representation [100] of explicit Runge–Kutta methods:

$$\begin{aligned} z_1 &= y_n, \\ z_i &= \sum_{j=1}^{i-1} (\alpha_{i,j} y_j + \beta_{i,j} h f(t_n + c_j h_n, z_j)), \\ y_{n+1} &= z_s. \end{aligned} \tag{2.10}$$

The coefficients of the Shu–Osher representation are not uniquely determined by the Butcher table [109]. In particular, the methods SSP(s,2), SSP(s,3), and SSP(10,4) implemented herein and presented in [73] have “almost” all zero coefficients appearing in  $\alpha_{i,i-1}$  and  $\beta_{i,i-1}$ . This feature facilitates their implementation in a low-storage manner. The corresponding coefficients and embedding weights can be found in [73] and [44], respectively.

## 2.7 MRISStep – Multirate infinitesimal step methods

The MRISStep time-stepping module in ARKODE is designed for IVPs of the form

$$\dot{y} = f^E(t, y) + f^I(t, y) + f^F(t, y), \quad y(t_0) = y_0. \tag{2.11}$$

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

- $f^E(t, y)$  contains the “slow-nonstiff” components of the system (this will be integrated using an explicit method and a large time step  $h_n^S$ ),
- $f^I(t, y)$  contains the “slow-stiff” components of the system (this will be integrated using an implicit method and a large time step  $h_n^S$ ), and
- $f^F(t, y)$  contains the “fast” components of the system (this will be integrated using a possibly different method than the slow time scale and a small time step  $h_n^F \ll h_n^S$ ).

As with ERKStep, MRISStep currently requires that problems be posed with an identity mass matrix,  $M(t) = I$ . The slow time scale may consist of only nonstiff terms ( $f^I \equiv 0$ ), only stiff terms ( $f^E \equiv 0$ ), or both nonstiff and stiff terms.

For cases with only a single slow right-hand side function (i.e.,  $f^E \equiv 0$  or  $f^I \equiv 0$ ), MRISStep provides multirate infinitesimal step (MIS) [96, 97, 98], first through fourth order multirate infinitesimal GARK (MRI-GARK) [93], and second through fifth order multirate exponential Runge–Kutta (MERK) [80] methods. For problems with an additively split slow right-hand side, MRISStep provides first through fourth order implicit-explicit MRI-GARK (IMEX-MRI-GARK) [30] and second through fourth order implicit-explicit multirate infinitesimal stage-restart (IMEX-MRI-SR) [46] methods. For a complete list of the methods available in MRISStep see §5.11.3.2. Additionally, users may supply their own method by defining and attaching a coupling table, see §5.11.3 for more information.

Generally, the slow (outer) method for each family derives from a single-rate method: MIS and MRI-GARK methods derive from explicit or diagonally-implicit Runge–Kutta methods, MERK methods derive from exponential Runge–Kutta methods, while IMEX-MRI-GARK and IMEX-MRI-SR methods derive from additive Runge–Kutta methods. In each case, the “infinitesimal” nature of the multirate methods derives from the fact that slow stages are computed by solving a set of auxiliary ODEs with a fast (inner) time integration method. Generally speaking, an  $s$ -stage method from of each family adheres to the following algorithm for a single step:

1. Set  $z_1 = y_{n-1}$ .
2. For  $i = 2, \dots, s$ , compute the stage solutions,  $z_i$ , by evolving the fast IVP

$$v'_i(t) = f^F(t, v_i) + r_i(t) \quad \text{for } t \in [t_{0,i}, t_{F,i}] \quad \text{with } v_i(t_{0,i}) = v_{0,i} \quad (2.12)$$

and setting  $z_i = v(t_{F,i})$ , and/or performing a standard explicit, diagonally-implicit, or additive Runge–Kutta stage update,

$$z_i - \theta_{i,i} h_n^S f^I(t_{n,i}^S, z_i) = a_i. \quad (2.13)$$

where  $t_{n,j}^S = t_{n-1} + h_n^S c_j^S$ .

3. Set  $y_n = z_s$ .
4. If the method has an embedding, compute the embedded solution,  $\tilde{y}$ , by evolving the fast IVP

$$\tilde{v}'(t) = f^F(t, \tilde{v}) + \tilde{r}(t) \quad \text{for } t \in [\tilde{t}_0, \tilde{t}_F] \quad \text{with } \tilde{v}(\tilde{t}_0) = \tilde{v}_0 \quad (2.14)$$

and setting  $\tilde{y}_n = \tilde{v}(\tilde{t}_F)$ , and/or performing a standard explicit, diagonally-implicit, or additive Runge–Kutta stage update,

$$\tilde{y}_n - \tilde{\theta} h_n^S f^I(t_n, \tilde{y}_n) = \tilde{a}. \quad (2.15)$$

Whether a fast IVP evolution or a stage update (or both) is needed depends on the method family (MRI-GARK, MERK, etc.). The specific aspects of the fast IVP forcing function ( $r_i(t)$  or  $\tilde{r}(t)$ ), the interval over which the IVP must be evolved ( $[t_{0,i}, t_{F,i}]$ ), the Runge–Kutta coefficients ( $\theta_{i,i}$  and  $\tilde{\theta}$ ), and the Runge–Kutta data ( $a_i$  and  $\tilde{a}$ ), are also determined by the method family. Generally, the forcing functions and data, are constructed using evaluations of the slow RHS functions,  $f^E$  and  $f^I$ , at preceding stages,  $z_j$ . The fast IVP solves can be carried out using any valid ARKODE integrator or a user-defined integration method (see section §5.11.4).

Below we summarize the details for each method family. For additional information, please see the references listed above.

## 2.7.1 MIS, MRI-GARK, and IMEX-MRI-GARK Methods

The methods in IMEX-MRI-GARK family, which includes MIS and MRI-GARK methods, are defined by a vector of slow stage time abscissae,  $c^S \in \mathbb{R}^s$ , and a set of coupling tensors,  $\Omega \in \mathbb{R}^{(s+1) \times s \times k}$  and  $\Gamma \in \mathbb{R}^{(s+1) \times s \times k}$ , that specify the slow-to-fast coupling for the explicit and implicit components, respectively.

The fast stage IVPs, (2.12), are evolved over non-overlapping intervals  $[t_{0,i}, t_{F,i}] = [t_{n,i-1}^S, t_{n,i}^S]$  with the initial condition  $v_{0,i} = z_{i-1}$ . The fast IVP forcing function is given by

$$r_i(t) = \frac{1}{\Delta c_i^S} \sum_{j=1}^{i-1} \omega_{i,j}(\tau) f^E(t_{n,j}^S, z_j) + \frac{1}{\Delta c_i^S} \sum_{j=1}^i \gamma_{i,j}(\tau) f^I(t_{n,j}^S, z_j)$$

where  $\Delta c_i^S = (c_i^S - c_{i-1}^S)$ ,  $\tau = (t - t_{n,i-1}^S) / (h_n^S \Delta c_i^S)$  is the normalized time, the coefficients  $\omega_{i,j}$  and  $\gamma_{i,j}$  are polynomials in time of degree  $k-1$  given by

$$\omega_{i,j}(\tau) = \sum_{\ell=1}^k \Omega_{i,j,\ell} \tau^{\ell-1} \quad \text{and} \quad \gamma_{i,j}(\tau) = \sum_{\ell=1}^k \Gamma_{i,j,\ell} \tau^{\ell-1}. \quad (2.16)$$

When the slow abscissa are repeated, i.e.  $\Delta c_i^S = 0$ , the fast IVP can be rescaled and integrated analytically leading to the Runge–Kutta update (2.13) instead of the fast IVP evolution. In this case the stage is computed as

$$z_i = z_{i-1} + h_n^S \sum_{j=1}^{i-1} \left( \sum_{\ell=1}^k \frac{\Omega_{i,j,\ell}}{\ell} \right) f^E(t_{n,j}^S, z_j) + h_n^S \sum_{j=1}^i \left( \sum_{\ell=1}^k \frac{\Gamma_{i,j,\ell}}{\ell} \right) f^I(t_{n,j}^S, z_j). \quad (2.17)$$

Similarly, the embedded solution IVP, (2.14), is evolved over the interval  $[\tilde{t}_0, \tilde{t}_F] = [t_{n,s-1}^S, t_n]$  with the initial condition  $\tilde{v}_0 = z_{s-1}$ .

As with standard ARK and DIRK methods, implicitness at the slow time scale is characterized by nonzero values on or above the diagonal of the  $k$  matrices in  $\Gamma$ . Typically, MRI-GARK and IMEX-MRI-GARK methods are at most diagonally-implicit (i.e.,  $\Gamma_{i,j,\ell} = 0$  for all  $\ell$  and  $j > i$ ). Furthermore, diagonally-implicit stages are characterized as being “solve-decoupled” if  $\Delta c_i^S = 0$  when  $\Gamma_{i,i,\ell} \neq 0$ , in which case the stage is computed as a standard ARK or DIRK update. Alternately, a diagonally-implicit stage  $i$  is considered “solve-coupled” if  $\Delta c_i^S \Gamma_{i,j,\ell} \neq 0$ , in which case the stage solution  $z_i$  is *both* an input to  $r_i(t)$  and the result of time-evolution of the fast IVP, necessitating an implicit solve that is coupled to the fast evolution. At present, only “solve-decoupled” diagonally-implicit MRI-GARK and IMEX-MRI-GARK methods are supported.

## 2.7.2 IMEX-MRI-SR Methods

The IMEX-MRI-SR family of methods perform *both* the fast IVP evolution, (2.12) or (2.14), *and* stage update, (2.13) or (2.15), in every stage (but these methods typically have far fewer stages than implicit MRI-GARK or IMEX-MRI-GARK methods). These methods are defined by a vector of slow stage time abscissae  $c^S \in \mathbb{R}^s$ , a set of coupling tensors  $\Omega \in \mathbb{R}^{(s+1) \times s \times k}$ , and a Butcher table of slow-implicit coefficients,  $\Gamma \in \mathbb{R}^{(s+1) \times s}$ .

The fast stage IVPs, (2.12), are evolved on overlapping intervals  $[t_{0,i}, t_{F,i}] = [t_{n-1}, t_{n,i}^S]$  with the initial condition  $v_{0,i} = y_{n-1}$ . The fast IVP forcing function is given by

$$r_i(t) = \frac{1}{c_i^S} \sum_{j=1}^{i-1} \omega_{i,j}(\tau) (f^E(t_{n,j}^S, z_j) + f^I(t_{n,j}^S, z_j)), \quad (2.18)$$

where  $\tau = (t - t_n)/(h_n^S c_i^S)$  is the normalized time, and the coefficients  $\omega_{i,j}$  are polynomials in time of degree  $k - 1$  that are also given by (2.16). The solution of these fast IVPs defines an intermediate stage solution,  $\tilde{z}_i$ .

The implicit solve that follows each fast IVP must solve the algebraic equation for  $z_i$

$$z_i = \tilde{z}_i + h_n^S \sum_{j=1}^i \gamma_{i,j} f^I(t_{n,j}^S, z_j). \quad (2.19)$$

We note that IMEX-MRI-SR methods are solve-decoupled by construction, and thus the structure of a given stage never needs to be deduced based on  $\Delta c_i^S$ . However, ARKODE still checks the value of  $\gamma_{i,i}$ , since if it zero then the stage update equation (2.19) simplifies to a simple explicit Runge–Kutta-like stage update.

The overall time step solution is given by the final internal stage solution, i.e.,  $y_n = z_s$ . The embedded solution is computing using the above algorithm for stage index  $s + 1$ , under the definition that  $c_{s+1}^S = 1$  (and thus the fast IVP portion is evolved over the full time step,  $[\tilde{t}_0, \tilde{t}_F] = [t_{n-1}, t_n]$ ).

## 2.7.3 MERK Methods

The MERK family of methods are only defined for multirate applications that are explicit at the slow time scale, i.e.,  $f^I = 0$ , but otherwise they are nearly identical to IMEX-MRI-SR methods. Specifically, like IMEX-MRI-SR methods, these evolve the fast IVPs (2.12) and (2.14) over the intervals  $[t_{0,i}, t_{F,i}] = [t_{n-1}, t_{n,i}^S]$  and  $[\tilde{t}_0, \tilde{t}_F] = [t_{n-1}, t_n]$ , respectively, and begin with the initial condition  $v_{0,i} = y_{n-1}$ . Furthermore, the fast IVP forcing functions are given by (2.18) with  $f^I = 0$ . As MERK-based applications lack the implicit slow operator, they do not require the solution of implicit algebraic equations.

However, unlike other MRI families, MERK methods were designed to admit a useful efficiency improvement. Since each fast IVP begins with the same initial condition,  $v_{0,i} = y_{n-1}$ , if multiple stages share the same forcing function  $r_i(t)$ , then they may be “grouped” together. This is achieved by sorting the final IVP solution time for each stage,  $t_{n,i}^S$ , and then evolving the inner solver to each of these stage times in order, storing the corresponding inner solver

solutions at these times as the stages  $z_i$ . For example, the ARKODE\_MERK54 method involves 11 stages, that may be combined into 5 distinct groups. The fourth group contains stages 7, 8, 9, and the embedding, corresponding to the  $c_i^S$  values  $7/10$ ,  $1/2$ ,  $2/3$ , and 1. Sorting these, a single fast IVP for this group must be evolved over the interval  $[t_{0,i}, t_{F,i}] = [t_{n-1}, t_n]$ , first pausing at  $t_{n-1} + \frac{1}{2}h_n^S$  to store  $z_8$ , then pausing at  $t_{n-1} + \frac{2}{3}h_n^S$  to store  $z_9$ , then pausing at  $t_{n-1} + \frac{7}{10}h_n^S$  to store  $z_7$ , and finally finishing the IVP solve to  $t_{n-1} + h_n^S$  to obtain  $\tilde{y}_n$ .

#### Note

Although all MERK methods were derived in [80] under an assumption that the fast function  $f^F(t, y)$  is linear in  $y$ , in [46] it was proven that MERK methods also satisfy all nonlinear order conditions up through their linear order. The lone exception is ARKODE\_MERK54, where it was only proven to satisfy all nonlinear conditions up to order 4 (since [46] did not establish the formulas for the order 5 conditions). All our numerical tests to date have shown ARKODE\_MERK54 to achieve fifth order for nonlinear problems, and so we conjecture that it also satisfies the nonlinear fifth order conditions.

## 2.8 SplittingStep – Operator splitting methods

The SplittingStep time-stepping module in ARKODE is designed for IVPs of the form

$$\dot{y} = f_1(t, y) + f_2(t, y) + \cdots + f_P(t, y), \quad y(t_0) = y_0,$$

with  $P > 1$  additive partitions. Operator splitting methods, such as those implemented in SplittingStep, allow each partition to be integrated separately, possibly with different numerical integrators or exact solution procedures. Coupling is only performed through initial conditions which are passed from the flow of one partition to the next.

The following algorithmic procedure is used in the Splitting-Step module:

1. For  $i = 1, \dots, r$  do:
  1. Set  $y_{n,i} = y_{n-1}$ .
  2. For  $j = 1, \dots, s$  do:
    1. For  $k = 1, \dots, P$  do:
      1. Let  $t_{\text{start}} = t_{n-1} + \beta_{i,j,k}h_n$  and  $t_{\text{end}} = t_{n-1} + \beta_{i,j+1,k}h_n$ .
      2. Let  $v(t_{\text{start}}) = y_{n,i}$ .
      3. For  $t \in [t_{\text{start}}, t_{\text{end}}]$  solve  $\dot{v} = f_k(t, v)$ .
      4. Set  $y_{n,i} = v(t_{\text{end}})$ .
2. Set  $y_n = \sum_{i=1}^r \alpha_i y_{n,i}$

Here,  $s$  denotes the number of stages, while  $r$  denotes the number of sequential methods within the overall operator splitting scheme. The sequential methods have independent flows which are linearly combined to produce the next step. The coefficients  $\alpha \in \mathbb{R}^r$  and  $\beta \in \mathbb{R}^{r \times (s+1) \times P}$  determine the particular scheme and properties such as the order of accuracy.

An alternative representation of the SplittingStep solution is

$$y_n = \sum_{i=1}^r \alpha_i \left( \phi_{\gamma_{i,s,P}h_n}^P \circ \phi_{\gamma_{i,s,P-1}h_n}^{P-1} \circ \cdots \circ \phi_{\gamma_{i,s,1}h_n}^1 \circ \phi_{\gamma_{i,s-1,P}h_n}^P \circ \cdots \circ \phi_{\gamma_{i,s-1,1}h_n}^1 \circ \cdots \circ \phi_{\gamma_{i,1,P}h_n}^P \circ \cdots \circ \phi_{\gamma_{i,1,1}h_n}^1 \right) (y_{n-1})$$

where  $\gamma_{i,j,k} = \beta_{i,j+1,k} - \beta_{i,j,k}$  is the scaling factor for the step size,  $h_n$ , and  $\phi_{h_n}^k$  is the flow map for partition  $k$ :

$$\phi_{h_n}^k(y_{n-1}) = v(t_n), \quad \begin{cases} v(t_{n-1}) = y_{n-1}, \\ \dot{v} = f_k(t, v). \end{cases}$$

For example, the Lie–Trotter splitting [18], given by

$$y_n = L_{h_n}(y_{n-1}) = (\phi_{h_n}^P \circ \phi_{h_n}^{P-1} \circ \cdots \circ \phi_{h_n}^1)(y_{n-1}), \quad (2.20)$$

is a first order, one-stage, sequential operator splitting method suitable for any number of partitions. Its coefficients are

$$\alpha_1 = 1, \\ \beta_{1,j,k} = \begin{cases} 0 & j = 1 \\ 1 & j = 2 \end{cases}, \quad j = 1, 2 \quad \text{and} \quad k = 1, \dots, P.$$

Higher order operator splitting methods are often constructed by composing the Lie–Trotter splitting with its adjoint:

$$L_{h_n}^* = L_{-h_n}^{-1} = \phi_{h_n}^1 \circ \phi_{h_n}^2 \circ \cdots \circ \phi_{h_n}^P. \quad (2.21)$$

This is the case for the Strang splitting [111]

$$y_n = S_{h_n}(y_{n-1}) = (L_{h_n/2}^* \circ L_{h_n/2})(y_{n-1}), \quad (2.22)$$

which has  $P$  stages and coefficients

$$\alpha_1 = 1, \\ \beta_{1,j,k} = \begin{cases} 0 & j = 1 \\ 1 & j + k > P + 1, \\ \frac{1}{2} & \text{otherwise} \end{cases}, \quad j = 1, \dots, P + 1 \quad \text{and} \quad k = 1, \dots, P.$$

SplittingStep provides standard operator splitting methods such as the Lie–Trotter and Strang splitting, as well as schemes of arbitrarily high order. Alternatively, users may provide their own coefficients (see §5.12.3). Generally, methods of order three and higher with real coefficients require backward integration, i.e., there exist negative  $\gamma_{i,j,k}$  coefficients. Currently, a fixed time step must be specified for the overall SplittingStep integrator, but partition integrators are free to use adaptive time steps.

## 2.9 SPRKStep – Symplectic Partitioned Runge–Kutta methods

The SPRKStep time-stepping module in ARKODE is designed for problems where the state vector is partitioned as

$$y(t) = \begin{bmatrix} p(t) \\ q(t) \end{bmatrix}$$

and the component partitioned IVP is given by

$$\begin{aligned} \dot{p} &= f_1(t, q), & p(t_0) &= p_0 \\ \dot{q} &= f_2(t, p), & q(t_0) &= q_0. \end{aligned} \quad (2.23)$$

The right-hand side functions  $f_1(t, p)$  and  $f_2(t, q)$  typically arise from the **separable** Hamiltonian system

$$H(t, p, q) = T(t, p) + V(t, q)$$

where

$$f_1(t, q) \equiv -\frac{\partial V(t, q)}{\partial q}, \quad f_2(t, p) \equiv \frac{\partial T(t, p)}{\partial p}.$$

When  $H$  is autonomous, then  $H$  is a conserved quantity. Often this corresponds to the conservation of energy (for example, in  $n$ -body problems). For non-autonomous  $H$ , the invariants are no longer directly obtainable from the Hamiltonian [112].



In practice, the ordering of the variables does not matter and is determined by the user. SPRKStep utilizes Symplectic Partitioned Runge-Kutta (SPRK) methods represented by the pair of explicit and diagonally implicit Butcher tableaux,

$$\begin{array}{c|cccc}
 c_1 & 0 & \cdots & 0 & 0 \\
 c_2 & a_1 & 0 & \cdots & \vdots \\
 \vdots & \vdots & \ddots & \ddots & \vdots \\
 c_s & a_1 & \cdots & a_{s-1} & 0 \\
 \hline
 & a_1 & \cdots & a_{s-1} & a_s
 \end{array}
 \quad
 \begin{array}{c|cccc}
 \hat{c}_1 & \hat{a}_1 & \cdots & 0 & 0 \\
 \hat{c}_2 & \hat{a}_1 & \hat{a}_2 & \cdots & \vdots \\
 \vdots & \vdots & \ddots & \ddots & \vdots \\
 \hat{c}_s & \hat{a}_1 & \hat{a}_2 & \cdots & \hat{a}_s \\
 \hline
 & \hat{a}_1 & \hat{a}_2 & \cdots & \hat{a}_s
 \end{array}$$

These methods approximately conserve a nearby Hamiltonian for exponentially long times [57]. SPRKStep makes the assumption that the Hamiltonian is separable, in which case the resulting method is explicit. SPRKStep provides schemes with order of accuracy and conservation equal to  $q = \{1, 2, 3, 4, 5, 6, 8, 10\}$ . The references for these methods and the default methods used are given in the section §19.4.

In the default case, the algorithm for a single time-step is as follows (for autonomous Hamiltonian systems the times provided to  $f_1$  and  $f_2$  can be ignored).

1. Set  $P_0 = p_{n-1}, Q_1 = q_{n-1}$
2. For  $i = 1, \dots, s$  do:
  1.  $P_i = P_{i-1} + h_n \hat{a}_i f_1(t_{n-1} + \hat{c}_i h_n, Q_i)$
  2.  $Q_{i+1} = Q_i + h_n a_i f_2(t_{n-1} + c_i h_n, P_i)$
3. Set  $p_n = P_s, q_n = Q_{s+1}$

Optionally, a different algorithm leveraging compensated summation can be used that is more robust to roundoff error at the expense of 2 extra vector operations per stage and an additional 5 per time step. It also requires one extra vector to be stored. However, it is significantly more robust to roundoff error accumulation [107]. When compensated summation is enabled, the following incremental form is used to compute a time step:

1. Set  $\Delta P_0 = 0, \Delta Q_1 = 0$
2. For  $i = 1, \dots, s$  do:
  1.  $\Delta P_i = \Delta P_{i-1} + h_n \hat{a}_i f_1(t_{n-1} + \hat{c}_i h_n, q_{n-1} + \Delta Q_i)$
  2.  $\Delta Q_{i+1} = \Delta Q_i + h_n a_i f_2(t_{n-1} + c_i h_n, p_{n-1} + \Delta P_i)$
3. Set  $\Delta p_n = \Delta P_s, \Delta q_n = \Delta Q_{s+1}$
4. Using compensated summation, set  $p_n = p_{n-1} + \Delta p_n, q_n = q_{n-1} + \Delta q_n$

Since temporal error based adaptive time-stepping is known to ruin the conservation property [57], SPRKStep requires that ARKODE be run using a fixed time-step size.

## 2.10 Error norms

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

$$\|v\|_{\text{WRMS}} = \left( \frac{1}{N} \sum_{i=1}^N (v_i w_i)^2 \right)^{1/2}. \quad (2.24)$$

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

$$w_i = (RTOL \cdot |y_{n-1,i}| + ATOL_i)^{-1}. \quad (2.25)$$



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

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

$$w_i = \left( RTOL \cdot |(M(t_{n-1}) y_{n-1})_i| + ATOL'_i \right)^{-1}, \quad (2.26)$$

where the user may specify a separate absolute residual tolerance value or array,  $ATOL'$ . The choice of weighting vector used in any given norm is determined by the quantity being measured: values having “solution” units use (2.25), whereas values having “equation” units use (2.26). Obviously, for problems with  $M = I$ , the solution and equation units are identical, in which case the solvers in ARKODE will use (2.25) when computing all error norms.

## 2.11 Time step adaptivity

A critical component of IVP “solvers” (rather than just time-steppers) is their adaptive control of local truncation error (LTE). At every step, we estimate the local error, and ensure that it satisfies tolerance conditions. If this local error test fails, then the step is recomputed with a reduced step size. To this end, the majority of the Runge–Kutta methods and many of the MRI methods in ARKODE admit an embedded solution  $\tilde{y}_n$ , as shown in equations (2.4), (2.8), and (2.14)–(2.15). Generally, these embedded solutions attain a slightly lower order of accuracy than the computed solution  $y_n$ . Denoting the order of accuracy for  $y_n$  as  $q$  and for  $\tilde{y}_n$  as  $p$ , most of these embedded methods satisfy  $p = q - 1$ . These values of  $q$  and  $p$  correspond to the *global* orders of accuracy for the method and embedding, hence each admit local truncation errors satisfying [55]

$$\begin{aligned} \|y_n - y(t_n)\| &= Ch_n^{q+1} + \mathcal{O}(h_n^{q+2}), \\ \|\tilde{y}_n - y(t_n)\| &= Dh_n^{p+1} + \mathcal{O}(h_n^{p+2}), \end{aligned} \quad (2.27)$$

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

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

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

$$T_n = \beta (y_n - \tilde{y}_n) = \beta h_n \sum_{i=1}^s \left[ (b_i^E - \tilde{b}_i^E) \hat{f}^E(t_{n,i}^E, z_i) + (b_i^I - \tilde{b}_i^I) \hat{f}^I(t_{n,i}^I, z_i) \right] \quad (2.28)$$

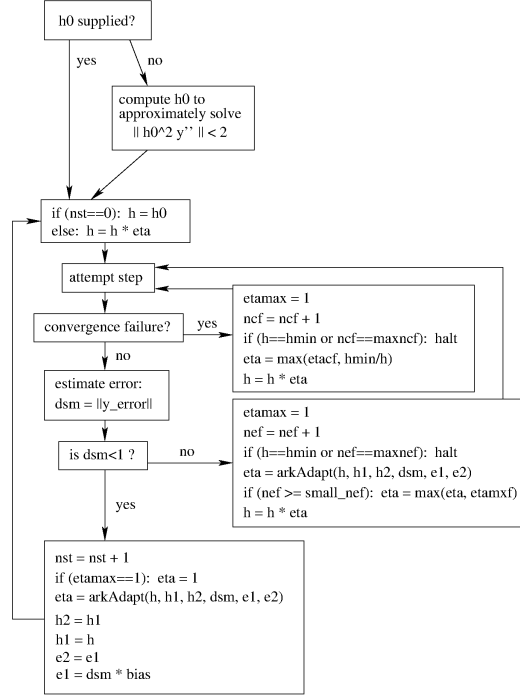
for ARK methods, and similarly for ERK methods. Here,  $\beta > 0$  is an error *bias* to help account for the error constant  $D$ ; the default value of this constant is  $\beta = 1.5$ , which may be modified by the user.

With this LTE estimate, the local error test is simply  $\|T_n\| < 1$  since this norm includes the user-specified tolerances. If this error test passes, the step is considered successful, and the estimate is subsequently used to determine the next step size, the algorithms used for this purpose are described in §2.11. If the error test fails, the step is rejected and a new step size  $h'$  is then computed using the same error controller as for successful steps. A new attempt at the step is made, and the error test is repeated. If the error test fails twice, then  $h'/h$  is limited above to 0.3, and limited below to 0.1 after an additional step failure. After seven error test failures, control is returned to the user with a failure message. We note that all of the constants listed above are only the default values; each may be modified by the user.

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

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

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



For some problems it may be preferable to avoid small step size adjustments. This can be especially true for problems that construct a Newton Jacobian matrix or a preconditioner for a nonlinear or an iterative linear solve, where this construction is computationally expensive, and where convergence can be seriously hindered through use of an inaccurate matrix. To accommodate these scenarios, the step is left unchanged when  $\eta \in [\eta_L, \eta_U]$ . The default values for this interval are  $\eta_L = 1$  and  $\eta_U = 1.0$  (so small step size adjustments are possible), and may be modified by the user.

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

The time-stepping modules in ARKODE adapt the step size in order to attain local errors within desired tolerances of the true solution. These adaptivity algorithms estimate the prospective step size  $h'$  based on the asymptotic local error estimates (2.27). We define the values  $\varepsilon_n$ ,  $\varepsilon_{n-1}$  and  $\varepsilon_{n-2}$  as

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

corresponding to the local error estimates for three consecutive steps,  $t_{n-3} \rightarrow t_{n-2} \rightarrow t_{n-1} \rightarrow t_n$ . These local error history values are all initialized to 1 upon program initialization, to accommodate the few initial time steps of a calculation where some of these error estimates have not yet been computed. With these estimates, ARKODE supports one of two approaches for temporal error control.

First, any valid implementation of the **SUNAdaptController** class §13.1 may be used by ARKODE’s adaptive time-stepping modules to provide a candidate error-based prospective step size  $h'$ .

Second, ARKODE’s adaptive time-stepping modules currently still allow the user to define their own time step adaptivity function,

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

allowing for problem-specific choices, or for continued experimentation with temporal error controllers. We note that this support has been deprecated in favor of the SUNAdaptController class, and will be removed in a future release.

### 2.11.1 Multirate time step adaptivity (MRISStep)

Since multirate applications evolve on multiple time scales, MRISStep supports additional forms of temporal adaptivity. Specifically, we consider time steps at two adjacent levels,  $h_n^S > h_n^F$ , where  $h_n^S$  is the step size used by MRISStep, and  $h_n^F$  is the step size used to solve the corresponding fast-time-scale IVPs in MRISStep, (2.12) and (2.14).

#### 2.11.1.1 Multirate temporal controls

We consider two categories of temporal controllers that may be used within MRI methods. The first (and simplest), are “decoupled” controllers, that consist of two separate single-rate temporal controllers: one that adapts the slow time scale step size,  $h_n^S$ , and the other that adapts the fast time scale step size,  $h_n^F$ . As these ignore any coupling between the two time scales, these methods should work well for multirate problems where the time scales are somewhat decoupled, and that errors introduced at one scale do not “pollute” the other.

The second category of controllers that we provide are  $h^S$ -Tol multirate controllers. The basic idea is that an adaptive time integration method will attempt to adapt step sizes to control the *local error* within each step to achieve a requested tolerance. However, MRI methods must ask an adaptive “inner” solver to produce the stage solutions  $v_i(t_{F,i})$  and  $\tilde{v}(\tilde{t}_F)$ , that result from sub-stepping over intervals  $[t_{0,i}, t_{F,i}]$  or  $[\tilde{t}_0, \tilde{t}_F]$ , respectively. Local errors within the inner integrator may accumulate, resulting in an overall inner solver error  $\varepsilon_n^F$  that exceeds the requested tolerance. If that inner solver can produce *both*  $v_i(t_{F,i})$  and an estimation of the accumulated error,  $\varepsilon_{n,approx}^F$ , then the tolerances provided to that inner solver can be adjusted accordingly to ensure stage solutions that are within the overall tolerances requested of the outer MRI method.

To this end, we assume that the inner solver will provide accumulated errors over each fast interval having the form

$$\varepsilon_n^F = c(t_n)h_n^S (\text{RTOL}_n^F), \quad (2.29)$$

where  $c(t)$  is independent of the tolerance or step size, but may vary in time. Single-scale adaptive controllers assume that the local error at a step  $n$  with step size  $h_n$  has order  $p$ , i.e.,

$$LTE_n = c(t_n)(h_n)^{p+1},$$

to predict candidate values  $h_{n+1}$ . We may therefore repurpose an existing single-scale controller to predict candidate values  $\text{RTOL}_{n+1}^F$  by supplying an “order”  $p = 0$  and a “control parameter”  $h_n = (\text{RTOL}_n^F)$ .

Thus to construct an  $h^S$ -Tol controller, we require three separate single-rate adaptivity controllers:

- **scontrol-H** – this is a single-rate controller that adapts  $h_n^S$  within the slow integrator to achieve user-requested solution tolerances.
- **scontrol-Tol** – this is a single-rate controller that adapts  $\text{RTOL}_n^F$  using the strategy described above.
- **fcontrol** – this adapts time steps  $h_n^F$  within the fast integrator to achieve the current tolerance,  $\text{RTOL}_n^F$ .

We note that both the decoupled and  $h^S$ -Tol controller families may be used in multirate calculations with an arbitrary number of time scales, since these focus on only one scale at a time, or on how a given time scale relates to the next-faster scale.

#### 2.11.1.2 Fast temporal error estimation

MRI temporal adaptivity requires estimation of the temporal errors that arise at *both* the slow and fast time scales, which we denote here as  $\varepsilon^S$  and  $\varepsilon^F$ , respectively. While the slow error may be estimated as  $\varepsilon^S = \|y_n - \tilde{y}_n\|$ , non-intrusive approaches for estimating  $\varepsilon^F$  are more challenging. ARKODE provides several strategies to help provide this

estimate, all of which assume the fast integrator is temporally adaptive and, at each of its  $m$  steps to reach  $t_n$ , computes an estimate of the local temporal error,  $\varepsilon_{n,m}^F$ . In this case, we assume that the fast integrator was run with the same absolute tolerances as the slow integrator, but that it may have used a potentially different relative solution tolerance,  $\text{RTOL}^F$ . The fast integrator then accumulates these local error estimates using either a “maximum accumulation” strategy,

$$\varepsilon_{max}^F = \text{RTOL}^F \max_{m \in \mathcal{S}} \|\varepsilon_{n,m}^F\|_{WRMS}, \quad (2.30)$$

an “additive accumulation” strategy,

$$\varepsilon_{sum}^F = \text{RTOL}^F \sum_{m \in \mathcal{S}} \|\varepsilon_{n,m}^F\|_{WRMS}, \quad (2.31)$$

or using an “averaged accumulation” strategy,

$$\varepsilon_{avg}^F = \frac{\text{RTOL}^F}{\Delta t_S} \sum_{m \in \mathcal{S}} h_{n,m} \|\varepsilon_{n,m}^F\|_{WRMS}, \quad (2.32)$$

where  $h_{n,m}$  is the step size that gave rise to  $\varepsilon_{n,m}^F$ ,  $\Delta t_S$  denotes the elapsed time over which  $\mathcal{S}$  is taken, and the norms are taken using the tolerance-informed error-weight vector. In each case, the sum or the maximum is taken over the set of all steps  $\mathcal{S}$  since the fast error accumulator has been reset.

## 2.12 Initial step size estimation

Before time step adaptivity can be accomplished, an initial step must be taken. These values may always be provided by the user; however, if these are not provided then ARKODE will estimate a suitable choice. Typically with adaptive methods, the first step should be chosen conservatively to ensure that it succeeds both in its internal solver algorithms, and its eventual temporal error test. However, if this initial step is too conservative then its computational cost will essentially be wasted. We thus strive to construct a conservative step that will succeed while also progressing toward the eventual solution.

Before commenting on the specifics of ARKODE, we first summarize two common approaches to initial step size selection. To this end, consider a simple single-time-scale ODE,

$$y'(t) = f(t, y), \quad y(t_0) = y_0 \quad (2.33)$$

For this, we may consider two Taylor series expansions of  $y(t_0 + h)$  around the initial time,

$$y(t_0 + h) = y_0 + hf(t_0, y_0) + \frac{h^2}{2} \frac{d}{dt} f(t_0 + \tau, y_0 + \eta), \quad (2.34)$$

and

$$y(t_0 + h) = y_0 + hf(t_0 + \tau, y_0 + \eta), \quad (2.35)$$

where  $t_0 + \tau$  is between  $t_0$  and  $t_0 + h$ , and  $y_0 + \eta$  is on the line segment connecting  $y_0$  and  $y(t_0 + h)$ .

Initial step size estimation based on the first-order Taylor expansion (2.34) typically attempts to determine a step size such that an explicit Euler method for (2.33) would be sufficiently accurate, i.e.,

$$\|y(t_0 + h_0) - (y_0 + h_0 f(t_0, y_0))\| \approx \left\| \frac{h^2}{2} \frac{d}{dt} f(t_0, y_0) \right\| < 1,$$

where we have assumed that  $y(t)$  is sufficiently differentiable, and that the norms include user-specified tolerances such that an error with norm less than one is deemed “acceptable.” Satisfying this inequality with a value of  $\frac{1}{2}$  and solving for  $h_0$ , we have

$$|h_0| = \frac{1}{\left\| \frac{d}{dt} f(t_0, y_0) \right\|^{1/2}}.$$

Finally, by estimating the time derivative with finite-differences,

$$\frac{d}{dt}f(t_0, y_0) \approx \frac{1}{\delta t} (f(t_0 + \delta t, y_0 + \delta t f(t_0, y_0)) - f(t_0, y_0)),$$

we obtain

$$|h_0| = \frac{\delta t^{1/2}}{\|f(t_0 + \delta t, y_0 + \delta t f(t_0, y_0)) - f(t_0, y_0)\|^{1/2}}. \quad (2.36)$$

Initial step size estimation based on the simpler Taylor expansion (2.35) instead assumes that the first calculated time step should be “close” to the initial state,

$$\|y(t_0 + h_0) - y_0\| \approx \|h_0 f(t_0, y_0)\| < 1,$$

where we again satisfy the inequality with a value of  $\frac{1}{2}$  to obtain

$$|h_0| = \frac{1}{2 \|f(t_0, y_0)\|}. \quad (2.37)$$

Comparing the two estimates (2.36) and (2.37), we see that the former has double the number of  $f$  evaluations, but that it has a less conservative estimate of  $h_0$ , particularly since we expect any valid time integration method to have at least  $\mathcal{O}(h)$  accuracy.

Of these two approaches, for calculations at a single time scale (e.g., using ARKStep), formula (2.36) is used, due to its more aggressive estimate for  $h_0$ .

### 2.12.1 Initial multirate step sizes

In MRI methods, initial time step selection is complicated by the fact that not only must an initial slow step size,  $h_0^S$ , be chosen, but a smaller initial step,  $h_0^F$ , must also be selected. Additionally, it is typically assumed that within MRI methods, evaluation of  $f^S$  is significantly more costly than evaluation of  $f^F$ , and thus we wish to construct these initial steps accordingly.

Under an assumption that conservative steps will be selected for both time scales, the error arising from temporal coupling between the slow and fast methods may be negligible. Thus, we estimate initial values of  $h_0^S$  and  $h_0^F$  independently. Due to our assumed higher cost of  $f^S$ , then for the slow time scale we employ the initial estimate (2.37) for  $h_0^S$  using  $f = f^S$ . Since the function  $f^F$  is assumed to be cheaper, we instead apply the estimate (2.36) for  $h_0^F$  using  $f = f^F$ , and enforce an upper bound  $|h_0^F| \leq \frac{|h_0^S|}{10}$ .

#### Note

If the fast integrator does not supply its “full RHS function”  $f^F$  for the MRI method to call, then we simply initialize  $h_0^F = \frac{h_0^S}{100}$ .

## 2.13 Explicit stability

For problems that involve a nonzero explicit component, i.e.  $f^E(t, y) \neq 0$  in ARKStep or for any problem in ERKStep, explicit and ImEx Runge–Kutta methods may benefit from additional user-supplied information regarding the explicit stability region. All ARKODE adaptivity methods utilize estimates of the local error, and it is often the case that such local error control will be sufficient for method stability, since unstable steps will typically exceed the error control tolerances. However, for problems in which  $f^E(t, y)$  includes even moderately stiff components, and especially for

higher-order integration methods, it may occur that a significant number of attempted steps will exceed the error tolerances. While these steps will automatically be recomputed, such trial-and-error can result in an unreasonable number of failed steps, increasing the cost of the computation. In these scenarios, a stability-based time step controller may also be useful.

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

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

where  $\Delta x$  is the spatial mesh size and  $\lambda$  is the fastest advective wave speed.

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

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

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

## 2.14 Fixed time stepping

While most of the time-stepping modules are designed for tolerance-based time step adaptivity, they additionally support a “fixed-step” mode. This mode is typically used for debugging purposes, for verification against hand-coded methods, or for problems where the time steps should be chosen based on other problem-specific information. In this mode, all internal time step adaptivity is disabled:

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

### Note

Since temporal error based adaptive time-stepping is known to ruin the conservation property of SPRK methods, SPRKStep employs a fixed time-step size by default.

### Note

Any methods that do not provide an embedding are required to be run in fixed-step mode.

Additional information on this mode is provided in the section [ARKODE Optional Inputs](#).

## 2.15 Algebraic solvers

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

### 2.15.1 Nonlinear solver methods

Methods with an implicit partition require solving implicit systems of the form

$$G(z_i) = 0. \quad (2.38)$$

In order to maximize solver efficiency, we define this root-finding problem differently based on the type of mass-matrix supplied by the user.

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

$$G(z_i) \equiv z_i - h_n A_{i,i}^I f^I(t_{n,i}^I, z_i) - a_i, \quad (2.39)$$

where we have the data

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

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

$$G(z_i) \equiv M z_i - h_n A_{i,i}^I f^I(t_{n,i}^I, z_i) - a_i, \quad (2.40)$$

where we have the data

$$a_i \equiv M y_{n-1} + h_n \sum_{j=1}^{i-1} [A_{i,j}^E f^E(t_{n,j}^E, z_j) + A_{i,j}^I f^I(t_{n,j}^I, z_j)].$$

#### Note

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

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

$$G(z_i) \equiv M(t_{n,i}^I)(z_i - a_i) - h_n A_{i,i}^I f^I(t_{n,i}^I, z_i) \quad (2.41)$$

where we have the data

$$a_i \equiv y_{n-1} + h_n \sum_{j=1}^{i-1} [A_{i,j}^E \hat{f}^E(t_{n,j}^E, z_j) + A_{i,j}^I \hat{f}^I(t_{n,j}^I, z_j)].$$

**Note**

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

- Similarly, in MRISep (that always assumes  $M = I$ ), MRI-GARK and IMEX-MRI-GARK methods have the residual

$$G(z_i) \equiv z_i - h_n^S \left( \sum_{k \geq 1} \frac{\Gamma_{i,i,k}}{k} \right) f^I(t_{n,i}^S, z_i) - a_i = 0 \quad (2.42)$$

where

$$a_i \equiv z_{i-1} + h_n^S \sum_{j=1}^{i-1} \left( \sum_{k \geq 1} \frac{\Gamma_{i,j,k}}{k} \right) f^I(t_{n,j}^S, z_j).$$

IMEX-MRI-SR methods have the residual

$$G(z_i) \equiv z_i - h_n^S \Gamma_{i,i} f^I(t_{n,i}^S, z_i) - a_i = 0 \quad (2.43)$$

where

$$a_i \equiv z_{i-1} + h_n^S \sum_{j=1}^{i-1} \Gamma_{i,j} f^I(t_{n,j}^S, z_j).$$

Upon solving for  $z_i$ , method stages must store  $f^E(t_{n,j}^E, z_i)$  and  $f^I(t_{n,j}^I, z_i)$ . It is possible to compute the latter without evaluating  $f^I$  after each nonlinear solve. Consider, for example, (2.39) which implies

$$f^I(t_{n,j}^I, z_i) = \frac{z_i - a_i}{h_n A_{i,i}^I} \quad (2.44)$$

when  $z_i$  is the exact root, and similar relations hold for non-identity mass matrices. This optimization can be enabled by `ARKodeSetDeduceImplicitRhs()` with the second argument in either function set to `SUNTRUE`. Another factor to consider when using this option is the amplification of errors from the nonlinear solver to the stages. In (2.44), nonlinear solver errors in  $z_i$  are scaled by  $1/(h_n A_{i,i}^I)$ . By evaluating  $f^I$  on  $z_i$ , errors are scaled roughly by the Lipschitz constant  $L$  of the problem. If  $h_n A_{i,i}^I L > 1$ , which is often the case when using implicit methods, it may be more accurate to use (2.44). Additional details are discussed in [99].

In each of the above nonlinear residual functions, if  $f^I(t, y)$  depends nonlinearly on  $y$  then (2.38) corresponds to a nonlinear system of equations; if instead  $f^I(t, y)$  depends linearly on  $y$  then this is a linear system of equations.

To solve each of the above root-finding problems ARKODE leverages `SUNNonlinearSolver` modules from the underlying SUNDIALS infrastructure (see section §11). By default, ARKODE selects a variant of Newton's method,

$$z_i^{(m+1)} = z_i^{(m)} + \delta^{(m+1)}, \quad (2.45)$$

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

$$\mathcal{A}(t_{n,i}^I, z_i^{(m)}) \delta^{(m+1)} = -G(z_i^{(m)}), \quad (2.46)$$



in which

$$\mathcal{A}(t, z) \approx M(t) - \gamma J(t, z), \quad J(t, z) = \frac{\partial f^I(t, z)}{\partial z}, \quad \text{and} \quad \gamma = h_n A_{i,i}^I \quad (2.47)$$

within ARKStep, or

$$\mathcal{A}(t, z) \approx I - \gamma J(t, z), \quad J(t, z) = \frac{\partial f^I(t, z)}{\partial z}, \quad \text{and} \quad \gamma = h_n^S \sum_{k \geq 1} \frac{\Gamma_{i,i,k}}{k} \quad (2.48)$$

within MRISStep.

In addition to Newton-based nonlinear solvers, the SUNDIALS SUNNonlinearSolver interface allows solvers of fixed-point type. These generally implement a fixed point iteration for solving an implicit stage  $z_i$ ,

$$z_i^{(m+1)} = \Phi \left( z_i^{(m)} \right) \equiv z_i^{(m)} - M(t_{n,i}^I)^{-1} G \left( z_i^{(m)} \right), \quad m = 0, 1, \dots \quad (2.49)$$

Unlike with Newton-based nonlinear solvers, fixed-point iterations generally *do not* require the solution of a linear system involving the Jacobian of  $f$  at each iteration.

Finally, if the user specifies that  $f^I(t, y)$  depends linearly on  $y$  in ARKStep or MRISStep and if the Newton-based SUNNonlinearSolver module is used, then the problem (2.38) will be solved using only a single Newton iteration. In this case, an additional user-supplied argument indicates whether this Jacobian is time-dependent or not, signaling whether the Jacobian or preconditioner needs to be recomputed at each stage or time step, or if it can be reused throughout the full simulation.

The optimal choice of solver (Newton vs fixed-point) is highly problem dependent. Since fixed-point solvers do not require the solution of linear systems involving the Jacobian of  $f$ , each iteration may be significantly less costly than their Newton counterparts. However, this can come at the cost of slower convergence (or even divergence) in comparison with Newton-like methods. While a Newton-based iteration is the default solver in ARKODE due to its increased robustness on very stiff problems, we strongly recommend that users also consider the fixed-point solver when attempting a new problem.

For either the Newton or fixed-point solvers, it is well-known that both the efficiency and robustness of the algorithm intimately depend on the choice of a good initial guess. The initial guess for these solvers is a prediction  $z_i^{(0)}$  that is computed explicitly from previously-computed data (e.g.  $y_{n-2}$ ,  $y_{n-1}$ , and  $z_j$  where  $j < i$ ). Additional information on the specific predictor algorithms is provided in section §2.15.5.

## 2.15.2 Linear solver methods

When a Newton-based method is chosen for solving each nonlinear system, a linear system of equations must be solved at each nonlinear iteration. For this solve ARKODE leverages another component of the shared SUNDIALS infrastructure, the “SUNLinearSolver,” described in section §10. These linear solver modules are grouped into two categories: matrix-based linear solvers and matrix-free iterative linear solvers. ARKODE’s interfaces for linear solves of these types are described in the subsections below.

### 2.15.2.1 Matrix-based linear solvers

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

$$\tilde{\mathcal{A}}(\tilde{t}, \tilde{z}) \delta^{(m+1)} = -G \left( z_i^{(m)} \right), \quad (2.50)$$

in which

$$\tilde{\mathcal{A}}(\tilde{t}, \tilde{z}) \approx M(\tilde{t}) - \tilde{\gamma} J(\tilde{t}, \tilde{z}), \quad \text{and} \quad \tilde{\gamma} = \tilde{h} A_{i,i}^I \quad (\text{ARKStep}) \quad (2.51)$$

or

$$\tilde{\mathcal{A}}(\tilde{t}, \tilde{z}) \approx I - \tilde{\gamma} J(\tilde{t}, \tilde{z}), \quad \text{and} \quad \tilde{\gamma} = \tilde{h} \sum_{k \geq 1} \frac{\Gamma_{i,i,k}}{k} \quad (\text{MRISStep}). \quad (2.52)$$

Here, the solution  $\tilde{z}$ , time  $\tilde{t}$ , and step size  $\tilde{h}$  upon which the modified equation rely, are merely values of these quantities from a previous iteration. In other words, the matrix  $\tilde{\mathcal{A}}$  is only computed rarely, and reused for repeated solves. As described below in section §2.15.2.3, the frequency at which  $\tilde{\mathcal{A}}$  is recomputed defaults to 20 time steps, but may be modified by the user.

When using the dense and band SUNMatrix objects for the linear systems (2.50), the Jacobian  $J$  may be supplied by a user routine, or approximated internally with finite-differences. In the case of differencing, we use the standard approximation

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

where  $e_j$  is the  $j$ -th unit vector, and the increments  $\sigma_j$  are given by

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

Here  $U$  is the unit roundoff,  $\sigma_0$  is a small dimensionless value, and  $w_j$  is the error weight defined in (2.25). 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 matrix bandwidth.

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

### 2.15.2.2 Matrix-free iterative linear solvers

In the case that a matrix-free iterative linear solver is chosen, an *inexact Newton iteration* is utilized. Here, the matrix  $\mathcal{A}$  is not itself constructed since the algorithms only require the product of this matrix with a given vector. Additionally, each Newton system (2.46) is not solved completely, since these linear solvers are iterative (hence the “inexact” in the name). As a result, for these linear solvers  $\mathcal{A}$  is applied in a matrix-free manner,

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

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

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

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

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

### 2.15.2.3 Updating the linear solver

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

- when starting the problem,

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

When an update of  $\tilde{\mathcal{A}}$  or  $P$  occurs, it 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 for the update. Reevaluating  $J$  (or instructing the user to update  $P$ ) occurs when:

- starting the problem,
- more than  $msbj = 50$  steps have been taken since the last evaluation (this value may be modified by the user),
- a convergence failure occurred with an outdated matrix, and the value  $\tilde{\gamma}$  of  $\gamma$  at the last update satisfies  $|\gamma/\tilde{\gamma} - 1| > 0.2$ ,
- a convergence failure occurred that forced a step size reduction, or
- if the problem is linearly implicit and  $\gamma$  has changed by a factor larger than 100 times machine epsilon.

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

As will be further discussed in section §2.15.4, in the case of most Krylov methods, preconditioning may be applied on the left, right, or on both sides of  $\mathcal{A}$ , with user-supplied routines for the preconditioner setup and solve operations.

## 2.15.3 Iteration Error Control

### 2.15.3.1 Nonlinear iteration error control

ARKODE provides a customized stopping test to the SUNNonlinearSolver module used for solving equation (2.38). This test is related to the temporal local error test, with the goal of keeping the nonlinear iteration errors from interfering with local error control. Denoting the final computed value of each stage solution as  $z_i^{(m)}$ , and the true stage solution solving (2.38) as  $z_i$ , we want to ensure that the iteration error  $z_i - z_i^{(m)}$  is “small” (recall that a norm less than 1 is already considered within an acceptable tolerance).

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

$$R_i \leftarrow \max \left\{ c_r R_i, \left\| \delta^{(m)} \right\| / \left\| \delta^{(m-1)} \right\| \right\}. \quad (2.53)$$

where the default factor  $c_r = 0.3$  is user-modifiable.

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

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

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

$$R_i \left\| \delta^{(m)} \right\| < \epsilon, \quad (2.54)$$

where the factor  $\epsilon$  has default value 0.1. We default to a maximum of 3 nonlinear iterations. We also declare the nonlinear iteration to be divergent if any of the ratios

$$\left\| \delta^{(m)} \right\| / \left\| \delta^{(m-1)} \right\| > r_{div}, \quad (2.55)$$

with  $m > 0$ , where  $r_{div}$  defaults to 2.3. If convergence fails in the nonlinear solver with  $\mathcal{A}$  current (i.e., not lagged), we reduce the step size  $h_n$  by a factor of  $\eta_{cf} = 0.25$ . The integration will be halted after  $max_{ncf} = 10$  convergence failures, or if a convergence failure occurs with  $h_n = h_{min}$ . However, since the nonlinearity of (2.38) may vary significantly based on the problem under consideration, these default constants may all be modified by the user.

### 2.15.3.2 Linear iteration error control

When a Krylov method is used to solve the linear Newton systems (2.46), its errors must also be controlled. To this end, we approximate the linear iteration error in the solution vector  $\delta^{(m)}$  using the preconditioned residual vector, e.g.  $r = P\mathcal{A}\delta^{(m)} + PG$  for the case of left preconditioning (the role of the preconditioner is further elaborated in the next section). In an attempt to ensure that the linear iteration errors do not interfere with the nonlinear solution error and local time integration error controls, we require that the norm of the preconditioned linear residual satisfies

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}. \quad (2.56)$$

Here  $\epsilon$  is the same value as that is used above for the nonlinear error control. The factor of 10 is used to ensure that the linear solver error does not adversely affect the nonlinear solver convergence. Smaller values for the parameter  $\epsilon_L$  are typically useful for strongly nonlinear or very stiff ODE systems, while easier ODE systems may benefit from a value closer to 1. The default value is  $\epsilon_L = 0.05$ , which may be modified by the user. We note that for linearly implicit problems the tolerance (2.56) is similarly used for the single Newton iteration.

### 2.15.4 Preconditioning

When using an inexact Newton method to solve the nonlinear system (2.38), an iterative method is used repeatedly to solve linear systems of the form  $\mathcal{A}x = b$ , where  $x$  is a correction vector and  $b$  is a residual vector. If this iterative method is one of the scaled preconditioned iterative linear solvers supplied with SUNDIALS, their efficiency may benefit tremendously from preconditioning. A system  $\mathcal{A}x = b$  can be preconditioned using any one of:

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

These Krylov iterative methods are then applied to a system with the matrix  $P^{-1}\mathcal{A}$ ,  $\mathcal{A}P^{-1}$ , or  $P_L^{-1}\mathcal{A}P_R^{-1}$ , instead of  $\mathcal{A}$ . In order to improve the convergence of the Krylov iteration, the preconditioner matrix  $P$ , or the product  $P_L P_R$  in the third case, should in some sense approximate the system matrix  $\mathcal{A}$ . Simultaneously, in order to be cost-effective the matrix  $P$  (or matrices  $P_L$  and  $P_R$ ) should be reasonably efficient to evaluate and solve. Finding an optimal point in this trade-off between rapid convergence and low cost can be quite challenging. Good choices are often problem-dependent (for example, see [23] for an extensive study of preconditioners for reaction-transport systems).

Most of the iterative linear solvers supplied with SUNDIALS allow for all three types of preconditioning (left, right or both), although for non-symmetric matrices  $\mathcal{A}$  we know of few situations where preconditioning on both sides is superior to preconditioning on one side only (with the product  $P = P_L P_R$ ). Moreover, for a given preconditioner matrix, the merits of left vs. right preconditioning are unclear in general, so we recommend that the user experiment with both choices. Performance can differ between these since the inverse of the left preconditioner is included in the

linear system residual whose norm is being tested in the Krylov algorithm. As a rule, however, if the preconditioner is the product of two matrices, we recommend that preconditioning be done either on the left only or the right only, rather than using one factor on each side. An exception to this rule is the PCG solver, that itself assumes a symmetric matrix  $\mathcal{A}$ , since the PCG algorithm in fact applies the single preconditioner matrix  $P$  in both left/right fashion as  $P^{-1/2}\mathcal{A}P^{-1/2}$ .

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

### 2.15.5 Implicit predictors

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

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

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

$$z_i^{(0)} = p_q(t_{n,i}^I). \quad (2.57)$$

As the stage times for MRI stages and implicit ARK and DIRK stages usually have non-negative abscissae (i.e.,  $c_j^I > 0$ ), it is typically the case that  $t_{n,j}^I$  (resp.,  $t_{n,j}^S$ ) is outside of the time interval containing the data used to construct  $p_q(t)$ , hence (2.57) will correspond to an extrapolant instead of an interpolant. The dangers of using a polynomial interpolant to extrapolate values outside the interpolation interval are well-known, with higher-order polynomials and predictions further outside the interval resulting in the greatest potential inaccuracies.

The prediction algorithms available in ARKODE therefore construct a variety of interpolants  $p_q(t)$ , having different polynomial order and using different interpolation data, to support “optimal” choices for different types of problems, as described below. We note that due to the structural similarities between implicit ARK and DIRK stages in ARKStep, and solve-decoupled implicit stages in MRISStep, we use the ARKStep notation throughout the remainder of this section, but each statement equally applies to MRISStep (unless otherwise noted).

#### 2.15.5.1 Trivial predictor

The so-called “trivial predictor” is given by the formula

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

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

### 2.15.5.2 Maximum order predictor

At the opposite end of the spectrum, ARKODE's interpolation modules discussed in section §2.2 can be used to construct a higher-order polynomial interpolant,  $p_q(t)$ . The implicit stage predictor is computed through evaluating the highest-degree-available interpolant at each stage time  $t_{n,i}^I$ .

### 2.15.5.3 Variable order predictor

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

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

### 2.15.5.4 Cutoff order predictor

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

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

### 2.15.5.5 Bootstrap predictor ( $M = I$ only) – deprecated

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

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

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

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

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

#### Note

This predictor has been deprecated, and will be removed from a future release.

### 2.15.5.6 Minimum correction predictor (ARKStep, $M = I$ only) – deprecated

The final predictor is not interpolation based; instead it utilizes all existing stage information from the current step to create a predictor containing all but the current stage solution. Specifically, as discussed in equations (2.4) and (2.38), 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

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

$$\Leftrightarrow$$

$$z_i = a_i.$$

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

#### Note

This predictor has been deprecated, and will be removed from a future release.

### 2.15.6 Mass matrix solver (ARKStep only)

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

$$b = Mv \tag{2.58}$$

or a linear solve

$$x = M^{-1}b \tag{2.59}$$

is required.

Of course, for problems in which  $M = I$  both of these operators are trivial. However for problems with non-identity mass matrix, these linear solves (2.59) may be handled using any valid SUNLinearSolver module, in the same manner as described in the section §2.15.2 for solving the linear Newton systems.

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

At present, for DIRK and ARK problems using a matrix-based solver for the Newton nonlinear iterations, the type of matrix (dense, band, sparse, or custom) for the Jacobian matrix  $J$  must match the type of mass matrix  $M$ , since these are combined to form the Newton system matrix  $\tilde{A}$ . When matrix-based methods are employed, the user must supply a routine to compute  $M(t)$  in the appropriate form to match the structure of  $\mathcal{A}$ , with a user-supplied routine of type [ARKLSMassFn\(\)](#). This matrix structure is used internally to perform any requisite mass matrix-vector products (2.58).



When matrix-free methods are selected, a routine must be supplied to perform the mass-matrix-vector product,  $Mv$ . As with iterative solvers for the Newton systems, preconditioning may be applied to aid in solution of the mass matrix systems (2.59). When using an iterative mass matrix linear solver, we require that the norm of the preconditioned linear residual satisfies

$$\|r\| \leq \epsilon_L \epsilon, \quad (2.60)$$

where again,  $\epsilon$  is the nonlinear solver tolerance parameter from (2.54). When using iterative system and mass matrix linear solvers,  $\epsilon_L$  may be specified separately for both tolerances (2.56) and (2.60).

In the algorithmic descriptions above there are five locations where a linear solve of the form (2.59) is required: (a) at each iteration of a fixed-point nonlinear solve, (b) in computing the Runge–Kutta right-hand side vectors  $\hat{f}_i^E$  and  $\hat{f}_i^I$ , (c) in constructing the time-evolved solution  $y_n$ , (d) in estimating the local temporal truncation error, and (e) in constructing predictors for the implicit solver iteration (see section §2.15.5.2). We note that different nonlinear solver approaches (i.e., Newton vs fixed-point) and different types of mass matrices (i.e., time-dependent versus fixed) result in different subsets of the above operations. We discuss each of these in the bullets below.

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

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

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

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

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

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

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

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

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

Similarly, we compute the local temporal error estimate  $T_n$  from equation (2.28) by solving systems of the form

$$M T_n = h \sum_{i=1}^s \left[ (b_i^E - \tilde{b}_i^E) f^E(t_{n,i}^E, z_i) + (b_i^I - \tilde{b}_i^I) f^I(t_{n,i}^I, z_i) \right]. \quad (2.61)$$

- For problems with either form of non-identity mass matrix, in constructing dense output and implicit predictors of degree 2 or higher (see the section §2.15.5.2 above), we compute the derivative information  $\hat{f}_k$  from the equation

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

In total, for problems with fixed mass matrix, we require only two mass-matrix linear solves (2.59) per attempted time step, with one more upon completion of a time step that meets the solution accuracy requirements. When fixed time-stepping is used ( $h_n = h$ ), the solve (2.61) is not performed at each attempted step.



Similarly, for problems with time-dependent mass matrix, we require  $2s$  mass-matrix linear solves (2.59) per attempted step, where  $s$  is the number of stages in the ARK method (only half of these are required for purely explicit or purely implicit problems, (2.5) or (2.6)), with one more upon completion of a time step that meets the solution accuracy requirements.

In addition to the above totals, when using a fixed-point nonlinear solver (assumed to require  $m$  iterations), we will need an additional  $ms$  mass-matrix linear solves (2.59) per attempted time step (but zero linear solves with the system Jacobian).

## 2.16 Rootfinding

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

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

The basic scheme used is to check for sign changes of any  $g_i(t)$  over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method [59]. In addition, each time  $g$  is evaluated, ARKODE checks to see if  $g_i(t) = 0$  exactly, and if so it reports this as a root. However, if an exact zero of any  $g_i$  is found at a point  $t$ , ARKODE computes  $g(t + \delta)$  for a small increment  $\delta$ , 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

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

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  $\alpha$  is a weight parameter. On the first two passes through the loop,  $\alpha$  is set to 1, making  $t_{mid}$  the secant method value. Thereafter,  $\alpha$  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,  $\alpha$  is set to 1. If the two sides were the same,  $\alpha$  is halved (if on the low side) or doubled (if on the high side). The value of  $t_{mid}$  is closer to  $t_{lo}$  when  $\alpha < 1$  and closer to  $t_{hi}$  when  $\alpha > 1$ . If the above value of  $t_{mid}$  is within  $\tau/2$  of  $t_{lo}$  or  $t_{hi}$ , it is adjusted inward, such

that its fractional distance from the endpoint (relative to the interval size) is between 0.1 and 0.5 (with 0.5 being the midpoint), and the actual distance from the endpoint is at least  $\tau/2$ .

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

## 2.17 Inequality Constraints

The ARKStep and ERKStep modules in ARKODE permit the user to impose optional inequality constraints on individual components of the solution vector  $y$ . Any of the following four constraints can be imposed:  $y_i > 0$ ,  $y_i < 0$ ,  $y_i \geq 0$ , or  $y_i \leq 0$ . The constraint satisfaction is tested after a successful step and before the error test. If any constraint fails, the step size is reduced and a flag is set to update the Jacobian or preconditioner if applicable. Rather than cutting the step size by some arbitrary factor, ARKODE estimates a new step size  $h'$  using a linear approximation of the components in  $y$  that failed the constraint test (including a safety factor of 0.9 to cover the strict inequality case). If a step fails to satisfy the constraints 10 times (a value which may be modified by the user) within a step attempt, or fails with the minimum step size, then the integration is halted and an error is returned. In this case the user may need to employ other strategies as discussed in §5.3.2 to satisfy the inequality constraints.

## 2.18 Relaxation Methods

When the solution of (2.1) is conservative or dissipative with respect to a smooth *convex* function  $\xi(y(t))$ , it is desirable to have the numerical method preserve these properties. That is  $\xi(y_n) = \xi(y_{n-1}) = \dots = \xi(y_0)$  for conservative systems and  $\xi(y_n) \leq \xi(y_{n-1})$  for dissipative systems. For examples of such problems, see the references below and the citations there in.

For such problems, ARKODE supports relaxation methods [66, 72, 85, 86] applied to ERK, DIRK, or ARK methods to ensure dissipation or preservation of the global function. The relaxed solution is given by

$$y_r = y_{n-1} + rd = ry_n + (1 - r)y_{n-1} \quad (2.62)$$

where  $d$  is the update to  $y_n$  (i.e.,  $h_n \sum_{i=1}^s (b_i^E \hat{f}_i^E + b_i^I \hat{f}_i^I)$  for ARKStep and  $h_n \sum_{i=1}^s b_i f_i$  for ERKStep) and  $r$  is the relaxation factor selected to ensure conservation or dissipation. Given an ERK, DIRK, or ARK method of at least second order with non-negative solution weights (i.e.,  $b_i \geq 0$  for ERKStep or  $b_i^E \geq 0$  and  $b_i^I \geq 0$  for ARKStep), the factor  $r$  is computed by solving the auxiliary scalar nonlinear system

$$F(r) = \xi(y_{n-1} + rd) - \xi(y_{n-1}) - re = 0 \quad (2.63)$$

at the end of each time step. The estimated change in  $\xi$  is given by  $e \equiv h_n \sum_{i=1}^s \langle \xi'(z_i), b_i^E f_i^E + b_i^I f_i^I \rangle$  where  $\xi'$  is the Jacobian of  $\xi$ .

Two iterative methods are provided for solving (2.63), Newton's method and Brent's method. When using Newton's method (the default), the iteration is halted either when the residual tolerance is met,  $F(r^{(k)}) < \epsilon_{\text{relax\_res}}$ , or when the difference between successive iterates satisfies the relative and absolute tolerances,  $|\delta_r^{(k)}| = |r^{(k)} - r^{(k-1)}| < \epsilon_{\text{relax\_rtol}} |r^{(k-1)}| + \epsilon_{\text{relax\_atol}}$ . Brent's method applies the same residual tolerance check and additionally halts when the bisection update satisfies the relative and absolute tolerances,  $|0.5(r_c - r^{(k)})| < \epsilon_{\text{relax\_rtol}} |r^{(k)}| + 0.5\epsilon_{\text{relax\_atol}}$  where  $r_c$  and  $r^{(k)}$  bound the root.

If the nonlinear solve fails to meet the specified tolerances within the maximum allowed number of iterations, the step size is reduced by the factor  $\eta_{\text{rf}}$  (default 0.25) and the step is repeated. Additionally, the solution of (2.63) should be  $r = 1 + \mathcal{O}(h_n^{q-1})$  for a method of order  $q$  [86]. As such, limits are imposed on the range of relaxation values allowed (i.e., limiting the maximum change in step size due to relaxation). A relaxation value greater than  $r_{\text{max}}$  (default 1.2) or

less than  $r_{\min}$  (default 0.8), is considered as a failed relaxation application and the step will be repeated with the step size reduced by  $\eta_{\text{rf}}$ .

For more information on utilizing relaxation Runge–Kutta methods, see §5.5.

## 2.19 Adjoint Sensitivity Analysis

Consider (2.7), but where the ODE also depends on some parameters,  $p$ , leading to the system

$$\dot{y} = f(t, y, p), \quad y(t_0) = y_0(p). \quad (2.64)$$

Now, suppose we have a functional  $g(t_f, y(t_f), p)$  for which we would like to compute the gradients

$$\frac{dg(t_f, y(t_f), p)}{dy}, \quad \text{and optionally,} \quad \frac{dg(t_f, y(t_f), p)}{dp}.$$

This most often arises in the form of an optimization problem such as

$$\min_{y(t_0), p} g(t_f, y(t_f), p). \quad (2.65)$$

The adjoint method is one approach to obtaining the gradients that is particularly efficient when there are relatively few functionals and a large number of parameters. While **CVODES** and **IDAS** provide *continuous* adjoint methods (differentiate-then-discretize), ARKODE provides *discrete* adjoint methods (discretize-then-differentiate). For the discrete adjoint approach, we first numerically discretize the original ODE (2.64). In the context of ARKODE, this is done with a one-step time integration scheme  $\varphi$  so that

$$y_0 = y(t_0), \quad y_n = \varphi(y_{n-1}). \quad (2.66)$$

Reformulating the optimization problem for the discrete case, we have

$$\min_{y_0, p} g(t_N, y_N, p), \quad (2.67)$$

where  $N$  is the index of the final time step so that  $t_N = t_f$  and  $y_N \approx y(t_f)$ . The gradients of (2.67) can be computed using the transposed chain rule backwards in time to obtain the discrete adjoint variables  $\lambda_N, \lambda_{N-1}, \dots, \lambda_0$  and  $\mu_N, \mu_{N-1}, \dots, \mu_0$ , where

$$\begin{aligned} \lambda_N &= g_y^*(t_N, y_N, p), \quad \lambda_n = \left( \frac{\partial \varphi}{\partial y_k}(y_n, p) \right)^* \lambda_{n+1} \\ \mu_N &= g_p^*(t_N, y_N, p), \quad \mu_n = \left( \frac{\partial \varphi}{\partial p}(y_n, p) \right)^* \lambda_{n+1}, \quad n = N-1, \dots, 0. \end{aligned} \quad (2.68)$$

### Warning

The CVODES and IDAS documentation uses  $\lambda$  to represent the adjoint variables needed to obtain the gradient  $dG/dp$  where  $G$  is an integral of  $g$ . Our use of  $\lambda$  in the following is akin to the use of  $\mu$  in the CVODES and IDAS docs.

The discrete adjoint variables represent the gradients of the discrete cost function

$$\frac{dg}{dy_0} = \lambda_0^*, \quad \frac{dg}{dp} = \mu_0^* + \lambda_0^* \left( \frac{\partial y_0}{\partial p} \right). \quad (2.69)$$

Given an s-stage explicit Runge–Kutta method (as in (2.8), but without the embedding), the discrete adjoint to compute  $\lambda_n$  and  $\mu_n$  starting from  $\lambda_{n+1}$  and  $\mu_{n+1}$  is given by

$$\begin{aligned}
 \Lambda_i &= h_n f_y^*(t_{n,i}, z_i, p) \left( b_i \lambda_{n+1} + \sum_{j=i+1}^s a_{j,i} \Lambda_j \right), \quad i = s, \dots, 1, \\
 \lambda_n &= \lambda_{n+1} + \sum_{j=1}^s \Lambda_j, \\
 \nu_i &= h_n f_p^*(t_{n,i}, z_i, p) \left( b_i \lambda_{n+1} + \sum_{j=i+1}^s a_{j,i} \Lambda_j \right), \\
 \mu_n &= \mu_{n+1} + \sum_{j=1}^s \nu_j.
 \end{aligned} \tag{2.70}$$

For more information on performing discrete adjoint sensitivity analysis using ARKODE see, §5.14. For a detailed derivation of the discrete adjoint methods see [54, 94]. See §15.1.1 for a brief discussion about the differences between the continuous and discrete adjoint methods, and why one would choose one over the other.

## Chapter 3

# Code 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 Fig. 3.1. The central integration modules, implemented in the files `arkode.h`, `arkode_impl.h`, `arkode_butcher.h`, `arkode.c`, `arkode_arkstep.c`, `arkode_erkstep.c`, `arkode_mrstep.c`, `arkode_sprkstep.c`, and `arkode_butcher.c`, deal with the evaluation of integration stages, the nonlinear solvers, estimation of the local truncation error, selection of step size, and interpolation to user output points, among other issues. ARKODE supports SUNNonlinearSolver modules in either root-finding or fixed-point form (see section §11) for any nonlinearly implicit problems that arise in computing each internal stage. When using Newton-based nonlinear solvers, 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; this/these are then invoked as needed during the integration process.

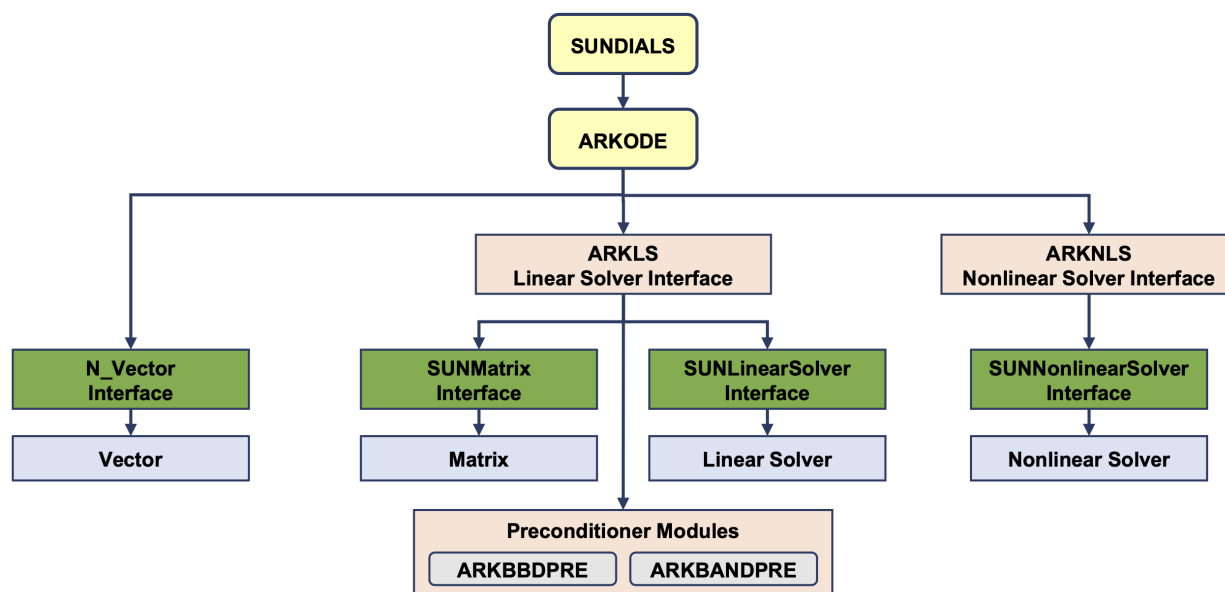


Fig. 3.1: *ARKODE organization*: Overall structure of the ARKODE package. Modules specific to ARKODE are the core infrastructure and timesteppers (ARKODE), linear solver interfaces (ARKLS), nonlinear solver interfaces (ARKNLS), and preconditioners (ARKBBDPRE and ARKBANDPRE); all other items correspond to generic SUNDIALS vector, matrix, and solver modules.

For solving these linear systems, ARKODE's linear solver interface supports both direct and iterative linear solvers adhering to the generic SUNLINSOL API (see §10). These solvers may utilize a SUNMATRIX object for storing Jacobian information, or they may be matrix-free. Since ARKODE can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to ARKODE will expand as new SUNLINSOL modules are developed.

For preconditioned iterative methods with either the system or mass matrix solves, the preconditioning must be supplied by the user in two phases: setup and solve. While there is no default choice of preconditioner for generic problems, the references [23] and [26], together with the example and demonstration programs included with ARKODE and CVODE, offer considerable assistance in building simple preconditioners.

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, C++ and Fortran 2003 applications there is no global data in the ARKODE package, and so in this respect it is reentrant. State information specific to the linear solver interface is saved in a separate data structure, a pointer to which resides in the ARKODE memory structure. State information specific to the linear solver implementation (and matrix implementation, if applicable) are stored in their own data structures, that are returned to the user upon construction, and subsequently provided to ARKODE for use.

## Chapter 4

# Getting Started

The packages that make up SUNDIALS are built upon shared classes for vectors, matrices, and algebraic solvers. In addition, the packages all leverage some other common infrastructure, which we discuss in this section.

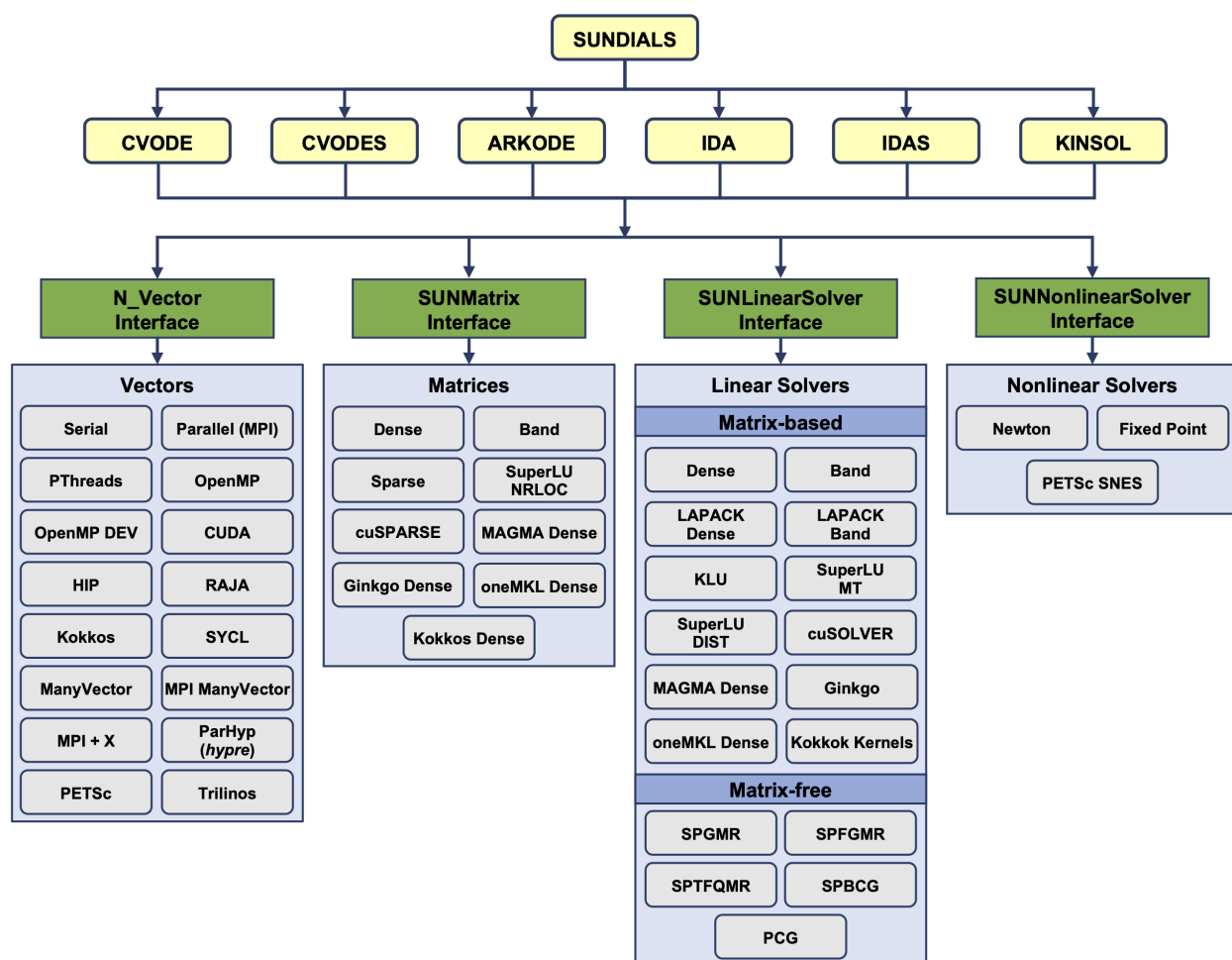


Fig. 4.1: High-level diagram of the SUNDIALS suite.

## 4.1 Data Types

SUNDIALS defines several data types in the header file `sundials_types.h`. These types are used in the SUNDIALS API and internally in SUNDIALS. It is not necessary to use these types in your application, but the type must be compatible with the SUNDIALS types in the API when calling SUNDIALS functions. The types that are defined are:

- `sunrealtype` – the floating-point type used by the SUNDIALS packages
- `sunindextype` – the integer type used for vector and matrix indices
- `suncountertype` – the integer type used for counter variables
- `sunbooleantype` – the type used for logic operations within SUNDIALS
- `SUNOutputFormat` – an enumerated type for SUNDIALS output formats
- `SUNComm` – a simple typedef to an `int` when SUNDIALS is built without MPI, or a `MPI_Comm` when built with MPI.

### 4.1.1 Floating point types

type `sunrealtype`

The type `sunrealtype` can be `float`, `double`, or `long double`, with the default being `double`. The user can change the precision of the arithmetic used in the SUNDIALS solvers at the configuration stage (see [SUNDIALS - PRECISION](#)).

Additionally, based on the current precision, `sundials_types.h` defines `SUN_BIG_REAL` to be the largest value representable as a `sunrealtype`, `SUN_SMALL_REAL` to be the smallest value representable as a `sunrealtype`, and `SUN_UNIT_ROUNDOFF` to be the difference between 1.0 and the minimum `sunrealtype` greater than 1.0.

Within SUNDIALS, real constants are set by way of a macro called `SUN_RCONST`. It is this macro that needs the ability to branch on the definition of `sunrealtype`. 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 `SUN_RCONST(1.0)` automatically expands to `1.0` if `sunrealtype` is `double`, to `1.0F` if `sunrealtype` is `float`, or to `1.0L` if `sunrealtype` is `long double`. SUNDIALS uses the `SUN_RCONST` macro internally to declare all of its floating-point constants.

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

A user program which uses the type `sunrealtype`, the `SUN_RCONST` macro, and the `SUNR` mathematical function macros is precision-independent except for any calls to precision-specific library functions. Our example programs use `sunrealtype`, `SUN_RCONST`, and the `SUNR` macros. Users can, however, use the type `double`, `float`, or `long double` in their code (assuming that this usage is consistent with the typedef for `sunrealtype`) and call the appropriate math library functions directly. Thus, a previously existing piece of C or C++ code can use SUNDIALS without modifying the code to use `sunrealtype`, `SUN_RCONST`, or the `SUNR` macros so long as the SUNDIALS libraries are built to use the corresponding precision (see §17.3).



### 4.1.2 Integer types used for indexing

#### type `sunindextype`

The type `sunindextype` is used for indexing array entries in SUNDIALS modules as well as for storing the total problem size (*e.g.*, vector lengths and matrix sizes). During configuration `sunindextype` may be selected to be either a 32- or 64-bit *signed* integer with the default being 64-bit (see [SUNDIALS\\_INDEX\\_SIZE](#)).

When using a 32-bit integer the total problem size is limited to  $2^{31} - 1$  and with 64-bit integers the limit is  $2^{63} - 1$ . For users with problem sizes that exceed the 64-bit limit an advanced configuration option is available to specify the type used for `sunindextype` (see [SUNDIALS\\_INDEX\\_TYPE](#)).

A user program which uses `sunindextype` to handle 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 compatible type (*e.g.*, `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 C or 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 §17.3).

### 4.1.3 Integer type used for counters

#### type `suncountertype`

The type `suncountertype` is used for counter variables in SUNDIALS (*e.g.*, number of stpes) and is the same as `long int`.

Added in version 7.3.0.

### 4.1.4 Boolean type

#### type `sunbooleantype`

As ANSI C89 (ISO C90) does not have a built-in boolean data type, SUNDIALS defines the type `sunbooleantype` as an `int`.

The advantage of using the name `sunbooleantype` (instead of `int`) is an increase in code readability. It also allows the programmer to make a distinction between `int` and boolean data. Variables of type `sunbooleantype` are intended to have only the two values: [SUNFALSE](#) or [SUNTRUE](#).

#### **SUNFALSE**

False (0)

#### **SUNTRUE**

True (1)

### 4.1.5 Output formatting type

#### enum `SUNOutputFormat`

The enumerated type [SUNOutputFormat](#) defines the enumeration constants for SUNDIALS output formats

#### enumerator `SUN_OUTPUTFORMAT_TABLE`

The output will be a table of values

enumerator **SUN\_OUTPUTFORMAT\_CSV**

The output will be a comma-separated list of key and value pairs e.g., `key1,value1,key2,value2,...`

**Note**

The Python module `tools/suntools` provides utilities to read and output the data from a SUNDIALS CSV output file using the key and value pair format.

## 4.1.6 MPI types

type **SUNComm**

A simple typedef to an *int* when SUNDIALS is built without MPI, or a `MPI_Comm` when built with MPI. This type exists solely to ensure SUNDIALS can support MPI and non-MPI builds.

**SUN\_COMM\_NULL**

A macro defined as `0` when SUNDIALS is built without MPI, or as `MPI_COMM_NULL` when built with MPI.

## 4.2 The SUNContext Type

Added in version 6.0.0.

All of the SUNDIALS objects (vectors, linear and nonlinear solvers, matrices, etc.) that collectively form a SUNDIALS simulation, hold a reference to a common simulation context object defined by the [SUNContext](#) class.

type **SUNContext**

An opaque pointer used by SUNDIALS objects for error handling, logging, profiling, etc.

Users should create a [SUNContext](#) object prior to any other calls to SUNDIALS library functions by calling:

[SUNErrCode](#) **SUNContext\_Create**([SUNComm](#) comm, [SUNContext](#) \*sunctx)

Creates a [SUNContext](#) object associated with the thread of execution. The data of the [SUNContext](#) class is private.

**Parameters**

- **comm** – the MPI communicator or `SUN_COMM_NULL` if not using MPI.
- **sunctx** – [in,out] upon successful exit, a pointer to the newly created [SUNContext](#) object.

**Returns**

[SUNErrCode](#) indicating success or failure.

The created [SUNContext](#) object should be provided to the constructor routines for different SUNDIALS classes/modules e.g.,

```
SUNContext sunctx;
void* package_mem;
N_Vector x;

SUNContext_Create(SUN_COMM_NULL, &sunctx);

package_mem = CVodeCreate(..., sunctx);
package_mem = IDACreate(..., sunctx);
```

(continues on next page)

(continued from previous page)

```
package_mem = KINCreate(..., sunctx);
package_mem = ARKStepCreate(..., sunctx);

x = N_VNew_<SomeVector>(..., sunctx);
```

After all other SUNDIALS code, the *SUNContext* object should be freed with a call to:

*SUNErrCode* **SUNContext\_Free**(*SUNContext* \*sunctx)

Frees the *SUNContext* object.

#### Parameters

- **sunctx** – pointer to a valid *SUNContext* object, NULL upon successful return.

#### Returns

*SUNErrCode* indicating success or failure.

#### Warning

When MPI is being used, the *SUNContext\_Free()* must be called prior to *MPI\_Finalize*.

The *SUNContext* API further consists of the following functions:

*SUNErrCode* **SUNContext\_GetLastError**(*SUNContext* sunctx)

Gets the last error code set by a SUNDIALS function call. The function then resets the last error code to *SUN\_SUCCESS*.

#### Parameters

- **sunctx** – a valid *SUNContext* object.

#### Returns

the last *SUNErrCode* recorded.

*SUNErrCode* **SUNContext\_PeekLastError**(*SUNContext* sunctx)

Gets the last error code set by a SUNDIALS function call. The function *does not* reset the last error code to *SUN\_SUCCESS*.

#### Parameters

- **sunctx** – a valid *SUNContext* object.

#### Returns

the last *SUNErrCode* recorded.

*SUNErrCode* **SUNContext\_PushErrorHandler**(*SUNContext* sunctx, *SUNErrorHandlerFn* err\_fn, void \*err\_user\_data)

Pushes a new *SUNErrorHandlerFn* onto the error handler stack so that it is called when an error occurs inside of SUNDIALS.

#### Parameters

- **sunctx** – a valid *SUNContext* object.
- **err\_fn** – a callback function of type *SUNErrorHandlerFn* to be pushed onto the error handler stack.
- **err\_user\_data** – a pointer that will be passed back to the callback function when it is called.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNContext\_PopErrorHandler**(*SUNContext* suncctx)

Pops the last *SUNErrorHandlerFn* off of the error handler stack.

**Parameters**

- **suncctx** – a valid *SUNContext* object.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNContext\_ClearErrHandlers**(*SUNContext* suncctx)

Clears the entire error handler stack. After doing this it is important to push an error handler onto the stack with *SUNContext\_PushErrorHandler* otherwise errors will be ignored.

**Parameters**

- **suncctx** – a valid *SUNContext* object.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNContext\_GetProfiler**(*SUNContext* suncctx, *SUNProfiler* \*profiler)

Gets the *SUNProfiler* object associated with the *SUNContext* object.

**Parameters**

- **suncctx** – a valid *SUNContext* object.
- **profiler** – [in,out] a pointer to the *SUNProfiler* object associated with this context; will be NULL if profiling is not enabled.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNContext\_SetProfiler**(*SUNContext* suncctx, *SUNProfiler* profiler)

Sets the *SUNProfiler* object associated with the *SUNContext* object.

**Parameters**

- **suncctx** – a valid *SUNContext* object.
- **profiler** – a *SUNProfiler* object to associate with this context; this is ignored if profiling is not enabled.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNContext\_SetLogger**(*SUNContext* suncctx, *SUNLogger* logger)

Sets the *SUNLogger* object associated with the *SUNContext* object.

**Parameters**

- **suncctx** – a valid *SUNContext* object.
- **logger** – a *SUNLogger* object to associate with this context; this is ignored if logging is not enabled.

**Returns**

*SUNErrCode* indicating success or failure.

Added in version 6.2.0.

*SUNErrCode* **SUNContext\_GetLogger**(*SUNContext* sunctx, *SUNLogger* \*logger)

Gets the *SUNLogger* object associated with the *SUNContext* object.

#### Parameters

- **sunctx** – a valid *SUNContext* object.
- **logger** – [in,out] a pointer to the *SUNLogger* object associated with this context; will be NULL if logging is not enabled.

#### Returns

*SUNErrCode* indicating success or failure.

Added in version 6.2.0.

### 4.2.1 Implications for task-based programming and multi-threading

Applications that need to have *concurrently initialized* SUNDIALS simulations need to take care to understand the following:

1. A *SUNContext* object must only be associated with *one* SUNDIALS simulation (a solver object and its associated vectors etc.) at a time.
  - Concurrently initialized is not the same as concurrently executing. Even if two SUNDIALS simulations execute sequentially, if both are initialized at the same time with the same *SUNContext*, behavior is undefined.
  - It is OK to reuse a *SUNContext* object with another SUNDIALS simulation after the first simulation has completed and all of the simulation's associated objects (vectors, matrices, algebraic solvers, etc.) have been destroyed.
2. The creation and destruction of a *SUNContext* object is cheap, especially in comparison to the cost of creating/destroying a SUNDIALS solver object.

The following (incomplete) code examples demonstrate these points using CVODE as the example SUNDIALS package.

```
SUNContext sunctxs[num_threads];
int ccode_initialized[num_threads];
void* ccode_mem[num_threads];

// Create
for (int i = 0; i < num_threads; i++) {
    sunctxs[i] = SUNContext_Create(...);
    ccode_mem[i] = CCodeCreate(..., sunctxs[i]);
    ccode_initialized[i] = 0; // not yet initialized
    // set optional ccode inputs...
}

// Solve
#pragma omp parallel for
for (int i = 0; i < num_problems; i++) {
    int retval = 0;
    int tid = omp_get_thread_num();
    if (!ccode_initialized[tid]) {
        retval = CCodeInit(ccode_mem[tid], ...);
        ccode_initialized[tid] = 1;
    }
}
```

(continues on next page)

(continued from previous page)

```

    } else {
        retval = CVodeReInit(cvode_mem[tid], ...);
    }
    CVode(cvode_mem[i], ...);
}

// Destroy
for (int i = 0; i < num_threads; i++) {
    // get optional ccode outputs...
    CVodeFree(&cvode_mem[i]);
    SUNContext_Free(&sunctxs[i]);
}

```

Since each thread has its own unique CVODE and SUNContext object pair, there should be no thread-safety issues. Users should be sure that you apply the same idea to the other SUNDIALS objects needed as well (e.g. an `N_Vector`).

The variation of the above code example demonstrates another possible approach:

```

// Create, Solve, Destroy
#pragma omp parallel for
for (int i = 0; i < num_problems; i++) {
    int retval = 0;
    void* cvode_mem;
    SUNContext sunctx;

    sunctx = SUNContext_Create(...);
    cvode_mem = CVodeCreate(..., sunctx);
    retval = CVodeInit(cvode_mem, ...);

    // set optional ccode inputs...

    CVode(cvode_mem, ...);

    // get optional ccode outputs...

    CVodeFree(&cvode_mem);
    SUNContext_Free(&sunctx);
}

```

So long as the overhead of creating/destroying the CVODE object is small compared to the cost of solving the ODE, this approach is a fine alternative to the first approach since `SUNContext_Create()` and `SUNContext_Free()` are much cheaper than the CVODE create/free routines.

## 4.2.2 Convenience class for C++ Users

For C++ users a RAII safe class, `sundials::Context`, is provided:

```

namespace sundials {

class Context : public sundials::ConvertibleTo<SUNContext>
{
public:

```

(continues on next page)

(continued from previous page)

```

explicit Context(SUNComm comm = SUN_COMM_NULL)
{
    sunctx_ = std::make_unique<SUNContext>();
    SUNContext_Create(comm, sunctx_.get());
}

/* disallow copy, but allow move construction */
Context(const Context&) = delete;
Context(Context&&)      = default;

/* disallow copy, but allow move operators */
Context& operator=(const Context&) = delete;
Context& operator=(Context&&) = default;

SUNContext get() override
{
    return *sunctx_.get();
}
SUNContext get() const override
{
    return *sunctx_.get();
}
operator SUNContext() override
{
    return *sunctx_.get();
}
operator SUNContext() const override
{
    return *sunctx_.get();
}

~Context()
{
    if (sunctx_) SUNContext_Free(sunctx_.get());
}

private:
std::unique_ptr<SUNContext> sunctx_;
};

} // namespace sundials

```

## 4.3 Error Checking

Added in version 7.0.0.

Until version 7.0.0, error reporting and handling was inconsistent throughout SUNDIALS. Starting with version 7.0.0 all of SUNDIALS (the core, implementations of core modules, and packages) reports error messages through the [SUNLogger](#) API. Furthermore, functions in the SUNDIALS core API (i.e., SUN or N\_V functions only) either return a [SUNErrCode](#), or (if they don't return a [SUNErrCode](#)) they internally record an error code (if an error occurs) within the [SUNContext](#) for the execution stream. This “last error” is accessible via the [SUNContext\\_GetLastError\(\)](#) or

`SUNContext_PeekLastError()` functions.

typedef int **SUNErrCode**

Thus, in user code, SUNDIALS core API functions can be checked for errors in one of two ways:

```
SUNContext sunctx;
SUNErrCode sunerr;
N_Vector v;
int length;
sunrealtype dotprod;

// Every code that uses SUNDIALS must create a SUNContext.
sunerr = SUNContext_Create(comm, &sunctx)
if (sunerr) { /* an error occurred, do something */ }

// Create a SUNDIALS serial vector.
// Some functions do not return an error code.
// We have to check for errors in these functions using SUNContext_GetLastError.
length = 2;
v = N_VNew_Serial(length, sunctx);
sunerr = SUNContext_GetLastError(sunctx);
if (sunerr) { /* an error occurred, do something */ }

// If the function returns a SUNErrCode, we can check it directly
sunerr = N_VLinearCombination(...);
if (sunerr) { /* an error occurred, do something */ }

// Another function that does not return a SUNErrCode.
dotprod = N_VDotProd(...);
SUNContext_GetLastError(sunctx);
if (sunerr) {
    /* an error occurred, do something */
} else {
    print("dotprod = %.2f\n", dotprod);
}
```

The function `SUNGetErrMsg()` can be used to get a message describing the error code.

const char \***SUNGetErrMsg**(*SUNErrCode* code)

Returns a message describing the error code.

**Parameters**

- **code** – the error code

**Returns**

a message describing the error code.

**Note**

It is recommended in most cases that users check for an error after calling SUNDIALS functions. However, users concerned with getting the most performance might choose to exclude or limit these checks.



**Warning**

If a function returns a *SUNErrCode* then the return value is the only place the error is available i.e., these functions do not store their error code as the “last error” so it is invalid to use *SUNContext\_GetLastError()* to check these functions for errors.

**4.3.1 Error Handler Functions**

When an error occurs in SUNDIALS, it calls error handler functions that have been pushed onto the error handler stack in last-in first-out order. Specific error handlers can be enabled by pushing them onto the error handler stack with the function *SUNContext\_PushErrorHandler()*. They may disabled by calling *SUNContext\_PopErrorHandler()* or *SUNContext\_ClearErrHandlers()*. A SUNDIALS error handler function has the type

```
typedef void (*SUNErrorHandlerFn)(int line, const char *func, const char *file, const char *msg, SUNErrCode err_code, void *err_user_data, SUNContext sunctx)
```

SUNDIALS provides a few different error handlers that can be used, or a custom one defined by the user can be provided (useful for linking SUNDIALS errors to your application’s error handling). The default error handler is *SUNLogErrorHandlerFn()* which logs an error to a specified file or `stderr` if no file is specified.

The error handlers provided in SUNDIALS are:

```
void SUNLogErrorHandlerFn(int line, const char *func, const char *file, const char *msg, SUNErrCode err_code, void *err_user_data, SUNContext sunctx)
```

Logs the error that occurred using the *SUNLogger* from *sunctx*. This is the default error handler.

**Parameters**

- **line** – the line number at which the error occurred
- **func** – the function in which the error occurred
- **file** – the file in which the error occurred
- **msg** – the message to log, if this is NULL then the default error message for the error code will be used
- **err\_code** – the error code for the error that occurred
- **err\_user\_data** – the user pointer provided to *SUNContext\_PushErrorHandler()*
- **sunctx** – pointer to a valid *SUNContext* object

**Returns**

void

```
void SUNAbortErrorHandlerFn(int line, const char *func, const char *file, const char *msg, SUNErrCode err_code, void *err_user_data, SUNContext sunctx)
```

Logs the error and aborts the program if an error occurred.

**Parameters**

- **line** – the line number at which the error occurred
- **func** – the function in which the error occurred
- **file** – the file in which the error occurred
- **msg** – this parameter is ignored
- **err\_code** – the error code for the error that occurred

- **err\_user\_data** – the user pointer provided to [SUNContext\\_PushErrorHandler\(\)](#)
- **sunctx** – pointer to a valid [SUNContext](#) object

**Returns**

void

void **SUNMPIAbortErrorHandlerFn**(int line, const char \*func, const char \*file, const char \*msg, [SUNErrCode](#) err\_code, void \*err\_user\_data, [SUNContext](#) sunctx)

Logs the error and calls `MPI_Abort` if an error occurred.

**Parameters**

- **line** – the line number at which the error occurred
- **func** – the function in which the error occurred
- **file** – the file in which the error occurred
- **msg** – this parameter is ignored
- **err\_code** – the error code for the error that occurred
- **err\_user\_data** – the user pointer provided to [SUNContext\\_PushErrorHandler\(\)](#)
- **sunctx** – pointer to a valid [SUNContext](#) object

**Returns**

void

## 4.4 Status and Error Logging

Added in version 6.2.0.

SUNDIALS includes a built-in logging functionality which can be used to direct error messages, warning messages, informational output, and debugging output to specified files. This capability requires enabling both build-time and run-time options to ensure the best possible performance is achieved.

### 4.4.1 Enabling Logging

To enable logging, the CMake option [SUNDIALS\\_LOGGING\\_LEVEL](#) must be set to the maximum desired output level when configuring SUNDIALS. See the [SUNDIALS\\_LOGGING\\_LEVEL](#) documentation for the numeric values corresponding to errors, warnings, info output, and debug output where errors < warnings < info output < debug output < extra debug output. By default only warning and error messages are logged.

When SUNDIALS is built with logging enabled, then the default logger (stored in the [SUNContext](#) object) may be configured through environment variables without any changes to user code. The available environment variables are:

```
SUNLOGGER_ERROR_FILENAME
SUNLOGGER_WARNING_FILENAME
SUNLOGGER_INFO_FILENAME
SUNLOGGER_DEBUG_FILENAME
```

These environment variables may be set to a filename string. There are two special filenames: `stdout` and `stderr`. These two filenames will result in output going to the standard output file and standard error file. For example, consider the CVODE Roberts example, where we can direct the informational output to the file `sun.log` as follows

```
SUNLOGGER_INFO_FILENAME=sun.log ./examples/cvode/serial/cvRoberts_dns
```

The different environment variables may all be set to the same file, or to distinct files, or some combination there of. To disable output for one of the streams, set the environment variable to an empty string. To leave the stream at its default output, do not set the environment variable. If [SUNDIALS\\_LOGGING\\_LEVEL](#) was set at build-time to a level lower than the corresponding environment variable, then setting the environment variable will do nothing. For example, if the logging level is set to 2 (errors and warnings), setting `SUNLOGGER_INFO_FILENAME` will do nothing.

Alternatively, the default logger can be accessed with [SUNContext\\_GetLogger\(\)](#) and configured using the [Logger API](#) or a user may create, configure, and attach a non-default logger using the [Logger API](#).

### Warning

A non-default logger should be created and attached to the context object prior to any other SUNDIALS calls in order to capture all log events.

The following examples demonstrate how to use the logging interface via the C API:

```
examples/arkode/CXX_serial/ark_analytic_sys.cpp
examples/cvode/serial/cvAdvDiff_bnd.c
examples/cvode/parallel/cvAdvDiff_diag_p.c
examples/kinsol/CXX_parallel/kin_em_p.cpp
examples/kinsol/CUDA_mpi/kin_em_mpicuda.cpp
```

## 4.4.2 Logging Output

Error or warning logs are a single line output with an error or warning message of the form

```
[level][rank][scope][label] message describing the error or warning
```

where the values in brackets have the following meaning:

- `level` is the log level of the message and will be `ERROR`, `WARNING`, `INFO`, or `DEBUG`
- `rank` is the MPI rank the message was written from (0 by default or if SUNDIALS was built without MPI enabled)
- `scope` is the message scope i.e., the name of the function from which the message was written
- `label` provides additional context or information about the logging output e.g., `begin-step`, `end-linear-solve`, etc.

Informational or debugging logs are either a single line output with a comma-separated list of key-value pairs of the form

```
[level][rank][scope][label] key1 = value1, key2 = value2
```

or multiline output with one value per line for keys corresponding to a vector or array e.g.,

```
[level][rank][scope][label] y(:) =
y[0]
y[1]
...
```

**Note**

When extra debugging output is enabled, the output will include vector values (so long as the `N_Vector` used supports printing). Depending on the problem size, this may result in very large logging files.

### 4.4.3 Logging Tools

Added in version 7.2.0.

To assist with extracting data from logging output files, the `tools` directory contains the `suntools` Python module which provides utilities for parsing log files in the `logs` sub-module.

`logs.log_file_to_list(filename)`

Parse a log file and return as a Python list of dictionaries.

This is the core parsing function that converts SUNDIALS log files into Python data structures. Use this function when you need to work with the data directly in Python (analysis, filtering, custom processing).

**Parameters**

**filename** (*str*) – The name of the log file to parse.

**Returns**

A list of dictionaries, one per step attempt.

**Return type**

list[dict]

The list returned for a time integrator log file will contain a dictionary for each step attempt:

```
[
  {
    "step": 1,
    "tn": 0.0,
    "h": 0.01,
    "status": "success",
    "dsm": 2.6e-13,
    "level": 0,
    "stages": [
      {"stage": 0, "implicit": 0, "tcur": 0.0, "status": "success"},
      {"stage": 1, "implicit": 0, "tcur": 0.001, "status": "success"},
      ...
    ],
    "compute-solution": {
      "mass type": 0,
      "status": "success"
    }
  },
  {
    "step": 2,
    "tn": 0.01,
    "h": 0.02,
    ...
  },
  ...
]
```

**Example usage:**

```

from suntools import logs
import matplotlib.pyplot as plt

# Parse the log file
data = logs.log_file_to_list("sun.log")

# Extract lists of passed and failed step data
passed = [s for s in data if "success" in s["status"]]
failed = [s for s in data if "failed" in s["status"]]

print("Steps stats: ")
print(f"  Attempted: {len(data)}")
print(f"  Successful: {len(passed)}")
print(f"  Failed:     {len(failed)}")
print(f"  Min Step:   {min(s[\"h\"] for s in passed)}")
print(f"  Max Step:   {max(s[\"h\"] for s in passed)}")
print(f"  Avg Step:   {sum(s[\"h\"] for s in passed) / len(passed)}")

# Plot step size history
s_steps, s_times, s_steps = logs.get_history(passed, "h")
f_steps, f_times, f_steps = logs.get_history(failed, "h")

fig, ax = plt.subplots()
ax.plot(s_times, s_steps, color="b", marker=".", label="passed")
ax.scatter(f_times, f_steps, color="r", marker="x", label="failed")
ax.legend(loc="best")
ax.set(xlabel="time", ylabel="step size", title="Step Size History")
plt.show()

```

`logs.print_log(log, indent=2)`

Print a log file list as formatted JSON.

This function takes parsed log data and prints it in a human-readable JSON format. Useful for debugging and quick inspection.

**Parameters**

- **log** (*list*) – The log file list from `log_file_to_list()`.
- **indent** (*int*) – The number of spaces to indent the JSON output (default: 2).

**Example usage:**

```

from suntools import logs

# Parse the log file
data = logs.log_file_to_list("sun.log")

# Print just the first few steps
logs.print_log(data[:5])

```

`logs.get_history(log, key, step_status=None, time_range=None, step_range=None, group_by_level=False)`

Extract the history of a key from a log file list created by `log_file_to_list()`.

**Parameters**

- **log** (*list*) – The log file list to extract values from.
- **key** (*str*) – The key to extract.
- **step\_status** (*str*) – Only extract values for steps which match the given status e.g., “success” or “failed”.
- **time\_range** (*[float, float]*) – Only extract values in the time interval, [low, high].
- **step\_range** (*[int, int]*) – Only extract values in the step number interval, [low, high].
- **group\_by\_level** (*bool*) – Group outputs by time level.

**Returns**

A list of steps, times, and values

The `tools` directory also contains example scripts demonstrating how to use the log parsing functions to extract and plot data.

- `log_example_print.py` – parses a log file and prints the log file list.
- `log_example.py` – plots the step size, order, or error estimate history from an ARKODE, CVODE(S), or IDA(S) log file.
- `log_example_mri.py` – plots the step size history from an ARKODE MRISep log file.

For more details on using an example script, run the script with the `--help` flag.

#### 4.4.4 Logger API

The central piece of the Logger API is the [\*SUNLogger\*](#) type:

type **SUNLogger**

An opaque pointer containing logging information.

When SUNDIALS is built with logging enabled, a default logging object is stored in the [\*SUNContext\*](#) object and can be accessed with a call to [\*SUNContext\\_GetLogger\(\)\*](#).

The enumerated type [\*SUNLogLevel\*](#) is used by some of the logging functions to identify the output level or file.

enum **SUNLogLevel**

The SUNDIALS logging level

enumerator **SUN\_LOGLEVEL\_ALL**

Represents all output levels

enumerator **SUN\_LOGLEVEL\_NONE**

Represents none of the output levels

enumerator **SUN\_LOGLEVEL\_ERROR**

Represents error-level logging messages

enumerator **SUN\_LOGLEVEL\_WARNING**

Represents warning-level logging messages

enumerator **SUN\_LOGLEVEL\_INFO**

Represents info-level logging messages

enumerator **SUN\_LOGLEVEL\_DEBUG**

Represents debug-level logging messages

The [\*SUNLogger\*](#) class provides the following methods.

*SUNErrCode* **SUNLogger\_Create**(*SUNComm* comm, int output\_rank, *SUNLogger* \*logger)

Creates a new *SUNLogger* object.

**Arguments:**

- comm – the MPI communicator to use, if MPI is enabled, otherwise can be SUN\_COMM\_NULL.
- output\_rank – the MPI rank used for output (can be -1 to print to all ranks).
- logger – [in,out] On input this is a pointer to a *SUNLogger*, on output it will point to a new *SUNLogger* instance.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_CreateFromEnv**(*SUNComm* comm, *SUNLogger* \*logger)

Creates a new *SUNLogger* object and opens the output streams/files from the environment variables:

```
SUNLOGGER_ERROR_FILENAME
SUNLOGGER_WARNING_FILENAME
SUNLOGGER_INFO_FILENAME
SUNLOGGER_DEBUG_FILENAME
```

**Arguments:**

- comm – the MPI communicator to use, if MPI is enabled, otherwise can be SUN\_COMM\_NULL.
- logger – [in,out] On input this is a pointer to a *SUNLogger*, on output it will point to a new *SUNLogger* instance.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetErrorFilename**(*SUNLogger* logger, const char \*error\_filename)

Sets the filename for error output.

**Arguments:**

- logger – a *SUNLogger* object.
- error\_filename – the name of the file to use for error output. Passing NULL or an empty string disables output for this stream.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetErrorFile**(*SUNLogger* logger, FILE \*error\_fp)

Sets the file pointer for error output.

The logger does not take ownership of error\_fp; the user is responsible for closing the file if needed. Passing NULL disables output for this stream.

**Arguments:**

- logger – a *SUNLogger* object.
- error\_fp – the FILE pointer to use for error output.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetWarningFilename**(*SUNLogger* logger, const char \*warning\_filename)

Sets the filename for warning output.

**Arguments:**

- logger – a *SUNLogger* object.
- warning\_filename – the name of the file to use for warning output. Passing NULL or an empty string disables output for this stream.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetWarningFile**(*SUNLogger* logger, FILE \*warning\_fp)

Sets the file pointer for warning output.

The logger does not take ownership of `warning_fp`; the user is responsible for closing the file if needed. Passing NULL disables output for this stream.

**Arguments:**

- logger – a *SUNLogger* object.
- warning\_fp – the FILE pointer to use for warning output.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetInfoFilename**(*SUNLogger* logger, const char \*info\_filename)

Sets the filename for info output.

**Arguments:**

- logger – a *SUNLogger* object.
- info\_filename – the name of the file to use for info output. Passing NULL or an empty string disables output for this stream.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetInfoFile**(*SUNLogger* logger, FILE \*info\_fp)

Sets the file pointer for info output.

The logger does not take ownership of `info_fp`; the user is responsible for closing the file if needed. Passing NULL disables output for this stream.

**Arguments:**

- logger – a *SUNLogger* object.
- info\_fp – the FILE pointer to use for info output.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetDebugFilename**(*SUNLogger* logger, const char \*debug\_filename)

Sets the filename for debug output.

**Arguments:**

- logger – a *SUNLogger* object.



- `debug_filename` – the name of the file to use for debug output. Passing an NULL or empty string disables output for this stream.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_SetDebugFile**(*SUNLogger* logger, FILE \*debug\_fp)

Sets the file pointer for debug output.

The logger does not take ownership of `debug_fp`; the user is responsible for closing the file if needed. Passing NULL disables output for this stream.

**Arguments:**

- `logger` – a *SUNLogger* object.
- `debug_fp` – the FILE pointer to use for debug output.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_QueueMsg**(*SUNLogger* logger, *SUNLogLevel* lvl, const char \*scope, const char \*label, const char \*msg\_txt, ...)

Queues a message to the output log level.

**Arguments:**

- `logger` – a *SUNLogger* object.
- `lvl` – the message log level (i.e. error, warning, info, debug).
- `scope` – the message scope (e.g. the function name).
- `label` – the message label.
- `msg_txt` – the message text itself.
- ... – the format string arguments

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_Flush**(*SUNLogger* logger, *SUNLogLevel* lvl)

Flush the message queue(s).

**Arguments:**

- `logger` – a *SUNLogger* object.
- `lvl` – the message log level (i.e. error, warning, info, debug or all).

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_GetOutputRank**(*SUNLogger* logger, int \*output\_rank)

Get the output MPI rank for the logger.

**Arguments:**

- `logger` – a *SUNLogger* object.
- `output_rank` – [in,out] On input this is a pointer to an int, on output it points to the int holding the output rank.

**Returns:**

- Returns zero if successful, or non-zero if an error occurred.

*SUNErrCode* **SUNLogger\_Destroy**(*SUNLogger* \*logger)

Free the memory for the *SUNLogger* object.

**Arguments:**

- logger – a pointer to the *SUNLogger* object.

**Returns:**

- Returns zero if successful, or non-zero if an error occur.

## 4.5 Performance Profiling

Added in version 6.0.0.

SUNDIALS includes a lightweight performance profiling layer that can be enabled at compile-time. Optionally, this profiling layer can leverage Caliper [19] for more advanced instrumentation and profiling. By default, only SUNDIALS library code is profiled. However, a public profiling API can be utilized to leverage the SUNDIALS profiler to time user code regions as well (see §4.5.2).

### 4.5.1 Enabling Profiling

To enable profiling, SUNDIALS must be built with the CMake option *SUNDIALS\_ENABLE\_PROFILING* set to ON. To utilize Caliper support, the CMake option *SUNDIALS\_ENABLE\_CALIPER* must also be set to ON. More details in regards to configuring SUNDIALS with CMake can be found in §17.

When SUNDIALS is built with profiling enabled and **without Caliper**, then the environment variable *SUNPROFILER\_PRINT* can be utilized to enable/disable the printing of profiler information. Setting *SUNPROFILER\_PRINT*=1 will cause the profiling information to be printed to stdout when the SUNDIALS simulation context is freed. Setting *SUNPROFILER\_PRINT*=0 will result in no profiling information being printed unless the *SUNProfiler\_Print()* function is called explicitly. By default, *SUNPROFILER\_PRINT* is assumed to be 0. *SUNPROFILER\_PRINT* can also be set to a file path where the output should be printed.

If Caliper is enabled, then users should refer to the [Caliper documentation](#) for information on getting profiler output. In most cases, this involves setting the *CALI\_CONFIG* environment variable.

**Note**

The SUNDIALS profiler requires POSIX timers or the Windows *profileapi.h* timers.

**Warning**

While the SUNDIALS profiling scheme is relatively lightweight, enabling profiling can still negatively impact performance. As such, it is recommended that profiling is enabled judiciously.

## 4.5.2 Profiler API

The primary way of interacting with the SUNDIALS profiler is through the following macros:

```
SUNDIALS_MARK_FUNCTION_BEGIN(profobj)
SUNDIALS_MARK_FUNCTION_END(profobj)
SUNDIALS_WRAP_STATEMENT(profobj, name, stmt)
SUNDIALS_MARK_BEGIN(profobj, name)
SUNDIALS_MARK_END(profobj, name)
```

Additionally, in C++ applications, the follow macro is available:

```
SUNDIALS_CXX_MARK_FUNCTION(profobj)
```

These macros can be used to time specific functions or code regions. When using the \*\_BEGIN macros, it is important that a matching \*\_END macro is placed at all exit points for the scope/function. The SUNDIALS\_CXX\_MARK\_FUNCTION macro only needs to be placed at the beginning of a function, and leverages RAII to implicitly end the region.

The `profobj` argument to the macro should be a `SUNProfiler` object, i.e.

type **SUNProfiler**

An opaque pointer containing profiling information.

When SUNDIALS is built with profiling, a default profiling object is stored in the `SUNContext` object and can be accessed with a call to `SUNContext_GetProfiler()`.

The `name` argument should be a unique string indicating the name of the region/function. It is important that the name given to the \*\_BEGIN macros matches the name given to the \*\_END macros.

In addition to the macros, the following methods of the `SUNProfiler` class are available.

int **SUNProfiler\_Create**(*SUNComm* comm, const char \*title, *SUNProfiler* \*p)

Creates a new `SUNProfiler` object.

### Arguments:

- `comm` – the MPI communicator to use, if MPI is enabled, otherwise can be `SUN_COMM_NULL`.
- `title` – a title or description of the profiler
- `p` – [in,out] On input this is a pointer to a `SUNProfiler`, on output it will point to a new `SUNProfiler` instance

### Returns:

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler\_Free**(*SUNProfiler* \*p)

Frees a `SUNProfiler` object.

### Arguments:

- `p` – [in,out] On input this is a pointer to a `SUNProfiler`, on output it will be `NULL`

### Returns:

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler\_Begin**(*SUNProfiler* p, const char \*name)

Starts timing the region indicated by the `name`.

### Arguments:

- `p` – a `SUNProfiler` object
- `name` – a name for the profiling region

**Returns:**

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler\_End**(*SUNProfiler* p, const char \*name)

Ends the timing of a region indicated by the `name`.

**Arguments:**

- `p` – a `SUNProfiler` object
- `name` – a name for the profiling region

**Returns:**

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler\_GetElapsedTime**(*SUNProfiler* p, const char \*name, double \*time)

Get the elapsed time for the timer “`name`” in seconds.

**Arguments:**

- `p` – a `SUNProfiler` object
- `name` – the name for the profiling region of interest
- `time` – upon return, the elapsed time for the timer

**Returns:**

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler\_GetTimerResolution**(*SUNProfiler* p, double \*resolution)

Get the timer resolution in seconds.

**Arguments:**

- `p` – a `SUNProfiler` object
- `resolution` – upon return, the resolution for the timer

**Returns:**

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler\_Print**(*SUNProfiler* p, FILE \*fp)

Prints out a profiling summary. When constructed with an MPI comm the summary will include the average and maximum time per rank (in seconds) spent in each marked up region.

**Arguments:**

- `p` – a `SUNProfiler` object
- `fp` – the file handler to print to

**Returns:**

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler\_Reset**(*SUNProfiler* p)

Resets the region timings and counters to zero.

**Arguments:**

- *p* – a SUNProfiler object

**Returns:**

- Returns zero if successful, or non-zero if an error occurred

### 4.5.3 Example Usage

The following is an excerpt from the CVODE example code `examples/cvode/serial/cvAdvDiff_bnd.c`. It is applicable to any of the SUNDIALS solver packages.

```
SUNContext ctx;
SUNProfiler profobj;

/* Create the SUNDIALS context */
retval = SUNContext_Create(SUN_COMM_NULL, &ctx);

/* Get a reference to the profiler */
retval = SUNContext_GetProfiler(ctx, &profobj);

/* ... */

SUNDIALS_MARK_BEGIN(profobj, "Integration loop");
umax = N_VMaxNorm(u);
PrintHeader(reltol, abstol, umax);
for(iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
    retval = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
    umax = N_VMaxNorm(u);
    retval = CVodeGetNumSteps(cvode_mem, &nst);
    PrintOutput(t, umax, nst);
}
SUNDIALS_MARK_END(profobj, "Integration loop");
PrintFinalStats(cvode_mem); /* Print some final statistics */
```

## 4.6 Getting Version Information

SUNDIALS provides additional utilities to all packages, that may be used to retrieve SUNDIALS version information at runtime.

int **SUNDIALSGetVersion**(char \*version, int len)

This routine fills a string with SUNDIALS version information.

**Arguments:**

- *version* – character array to hold the SUNDIALS version information.
- *len* – allocated length of the *version* character array.

**Return value:**

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS version

**Notes:**

An array of 25 characters should be sufficient to hold the version information.

int **SUNDIALSGetVersionNumber**(int \*major, int \*minor, int \*patch, char \*label, int len)

This routine sets integers for the SUNDIALS major, minor, and patch release numbers and fills a string with the release label if applicable.

**Arguments:**

- *major* – SUNDIALS release major version number.
- *minor* – SUNDIALS release minor version number.
- *patch* – SUNDIALS release patch version number.
- *label* – string to hold the SUNDIALS release label.
- *len* – allocated length of the *label* character array.

**Return value:**

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS label

**Notes:**

An array of 10 characters should be sufficient to hold the label information. If a label is not used in the release version, no information is copied to *label*.

## 4.7 Features for GPU Accelerated Computing

In this section, we introduce the SUNDIALS GPU programming model and highlight SUNDIALS GPU features. The model leverages the fact that all of the SUNDIALS packages interact with simulation data either through the shared vector, matrix, and solver APIs or through user-supplied callback functions. Thus, under the model, the overall structure of the user's calling program, and the way users interact with the SUNDIALS packages is similar to using SUNDIALS in CPU-only environments.

### 4.7.1 SUNDIALS GPU Programming Model

As described in [15], within the SUNDIALS GPU programming model, all control logic executes on the CPU, and all simulation data resides wherever the vector or matrix object dictates as long as SUNDIALS is in control of the program. That is, SUNDIALS will not migrate data (explicitly) from one memory space to another. Except in the most advanced use cases, it is safe to assume that data is kept resident in the GPU-device memory space. The consequence of this is that, when control is passed from the user's calling program to SUNDIALS, simulation data in vector or matrix objects must be up-to-date in the device memory space. Similarly, when control is passed from SUNDIALS to the user's calling program, the user should assume that any simulation data in vector and matrix objects are up-to-date in the device memory space. To put it succinctly, *it is the responsibility of the user's calling program to manage data coherency between the CPU and GPU-device memory spaces* unless unified virtual memory (UVM), also known as managed memory, is being utilized. Typically, the GPU-enabled SUNDIALS modules provide functions to copy data from the host to the device and vice-versa as well as support for unmanaged memory or UVM. In practical terms, the way SUNDIALS handles distinct host and device memory spaces means that *users need to ensure that the user-supplied functions, e.g. the right-hand side function, only operate on simulation data in the device memory space* otherwise extra memory transfers will be required and performance will suffer. The exception to this rule is if some form of hybrid data partitioning (achievable with the NVECTOR\_MANYVECTOR, see §8.17) is utilized.

SUNDIALS provides many native shared features and modules that are GPU-enabled. Currently, these include the NVIDIA CUDA platform [5], AMD ROCm/HIP [2], and Intel oneAPI [3]. Table 4.1–Table 4.4 summarize the shared SUNDIALS modules that are GPU-enabled, what GPU programming environments they support, and what class of memory they support (unmanaged or UVM). Users may also supply their own GPU-enabled *N\_Vector*, *SUNMatrix*,

*SUNLinearSolver*, or *SUNNonlinearSolver* implementation, and the capabilities will be leveraged since SUNDIALS operates on data through these APIs.

In addition, SUNDIALS provides a memory management helper module (see §16) to support applications which implement their own memory management or memory pooling.

Table 4.1: List of SUNDIALS GPU-enabled N\_Vector Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
<i>NVECTOR_CUDA</i>	X			X	X
<i>NVECTOR_HIP</i>	X	X		X	X
<i>NVECTOR_SYCL</i>	X <sup>3</sup>	X <sup>3</sup>	X	X	X
<i>NVECTOR_RAJA</i>	X	X	X	X	X
<i>NVECTOR_KOKKOS</i>	X	X	X	X	X
<i>NVECTOR_OPENMPDEV</i>	X	X <sup>2</sup>	X <sup>2</sup>	X	

Table 4.2: List of SUNDIALS GPU-enabled SUNMatrix Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
<i>SUNMATRIX_CUSPARSE</i>	X			X	X
<i>SUNMATRIX_ONEMKLDENSE</i>	X <sup>3</sup>	X <sup>3</sup>	X	X	X
<i>SUNMATRIX_MAGMADENSE</i>	X	X		X	X
<i>SUNMATRIX_GINKGO</i>	X	X		X	X
<i>SUNMATRIX_KOKKOSDENSE</i>	X	X		X	X

Table 4.3: List of SUNDIALS GPU-enabled SUNLinearSolver Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
<i>SUNLINSOL_CUSOLVERSP</i>	X			X	X
<i>SUNLINSOL_ONEMKLDENSE</i>	X <sup>3</sup>	X <sup>3</sup>	X	X	X
<i>SUNLINSOL_MAGMADENSE</i>	X			X	X
<i>SUNLINSOL_GINKGO</i>	X	X		X	X
<i>SUNLINSOL_KOKKOSDENSE</i>	X	X		X	X
<i>SUNLINSOL_SPGMR</i>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>
<i>SUNLINSOL_SPGFMR</i>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>
<i>SUNLINSOL_SPTFQMR</i>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>
<i>SUNLINSOL_SPBCGS</i>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>
<i>SUNLINSOL_PCG</i>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>

Table 4.4: List of SUNDIALS GPU-enabled SUNNonlinearSolver Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
<i>SUNNONLINSOL_NEWTON</i>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>
<i>SUNNONLINSOL_FIXEDPOINT</i>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>

Notes regarding the above tables:

1. This module inherits support from the NVECTOR module used
2. Support for ROCm/HIP and oneAPI are currently untested.
3. Support for CUDA and ROCm/HIP are currently untested.

In addition, note that implicit UVM (i.e. `malloc` returning UVM) is not accounted for.

#### 4.7.2 Steps for Using GPU Accelerated SUNDIALS

For any SUNDIALS package, the generalized steps a user needs to take to use GPU accelerated SUNDIALS are:

1. Utilize a GPU-enabled `N_Vector` implementation. Initial data can be loaded on the host, but must be in the device memory space prior to handing control to SUNDIALS.
2. Utilize a GPU-enabled `SUNLinearSolver` linear solver (if applicable).
3. Utilize a GPU-enabled `SUNMatrix` implementation (if using a matrix-based linear solver).
4. Utilize a GPU-enabled `SUNNonlinearSolver` nonlinear solver (if applicable).
5. Write user-supplied functions so that they use data only in the device memory space (again, unless an atypical data partitioning is used). A few examples of these functions are the right-hand side evaluation function, the Jacobian evaluation function, or the preconditioner evaluation function. In the context of CUDA and the right-hand side function, one way a user might ensure data is accessed on the device is, for example, calling a CUDA kernel, which does all of the computation, from a CPU function which simply extracts the underlying device data array from the `N_Vector` object that is passed from SUNDIALS to the user-supplied function.

Users should refer to the above tables for a complete list of GPU-enabled native SUNDIALS modules.



## Chapter 5

# Using ARKODE

This chapter discusses usage of ARKODE for the solution of initial value problems (IVPs) in C, C++ and Fortran applications. The chapter builds upon §4. Unlike other packages in SUNDIALS, ARKODE provides an infrastructure for one-step methods. However, ARKODE’s individual time-stepping methods, including definition of the IVP itself, are handled by time-stepping modules that sit on top of ARKODE. While most of the routines to use ARKODE generally apply to all of its time-stepping modules, some of these apply to only a subset of these “steppers,” while others are specific to a given stepper.

Thus, we organize this chapter as follows. We first discuss commonalities to each of ARKODE’s time-stepping modules. These commonalities include the locations and naming conventions for the library and header files, data types in SUNDIALS, the layout of the user’s main program, and a variety of user-callable and user-supplied functions. For these user-callable routines, we distinguish those that apply for only a subset of ARKODE’s time-stepping modules. We then describe shared utilities that are supported by some of ARKODE’s time stepping modules, including “relaxation” methods and preconditioners. Following our discussion of these commonalities, we separately discuss the usage details that are specific to each of ARKODE’s time stepping modules: *ARKStep*, *ERKStep*, *ForcingStep*, *LSRKStep*, *MRIStep*, *SplittingStep*, and *SPRKStep*.

ARKODE also uses various input and output constants; these are defined as needed throughout this chapter, but for convenience the full list is provided separately in §18.

The example programs for ARKODE are located in the source code `examples/arkode` folder. We note that these may be helpful as templates for new codes. Users with applications written in Fortran should see the chapter §20, which describes the Fortran interfaces for SUNDIALS, and we additionally include multiple Fortran example programs in the ARKODE `examples` directory.

When solving problems with an implicit component, we note that not all SUNLINSOL, SUNMATRIX, and preconditioning modules are compatible with all NVECTOR implementations. Details on compatibility are given in the documentation for each SUNMATRIX (see §9) and each SUNLINSOL module (see §10). For example, NVECTOR\_PARALLEL is not compatible with the dense, banded, or sparse SUNMATRIX types, or with the corresponding dense, banded, or sparse SUNLINSOL modules. Please check §9 and §10 to verify compatibility between these modules. In addition to that documentation, we note that the ARKBANDPRE preconditioning module is only compatible with the NVECTOR\_SERIAL, NVECTOR\_OPENMP or NVECTOR\_PTHREADS vector implementations, and the preconditioner module ARKBBDPRE can only be used with NVECTOR\_PARALLEL.

### 5.1 Access to library and header files

At this point, it is assumed that the installation of ARKODE, following the procedure described in §17, has been completed successfully. In the proceeding text, the directories `libdir` and `incdir` are the installation library and in-

clude directories, respectively. For a default installation, these are `instdir/lib` and `instdir/include`, respectively, where `instdir` is the directory where SUNDIALS was installed.

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. ARKODE symbols are found in `libdir/libsundials_arkode.lib`. Thus, in addition to linking to `libdir/libsundials_core.lib`, ARKODE users need to link to the ARKODE library. Symbols for additional SUNDIALS modules, vectors and algebraic solvers, are found in

```
<libdir>/libsundials_nvec*.lib
<libdir>/libsundials_sunmat*.lib
<libdir>/libsundials_sunlinsol*.lib
<libdir>/libsundials_sunnonlinsol*.lib
<libdir>/libsundials_sunmem*.lib
```

The file extension `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The relevant header files for ARKODE are located in the subdirectories `incdir/include/arkode`. To use ARKODE the application needs to include the header file(s) for the ARKODE time-stepper(s) of choice in addition to the SUNDIALS core header file.

```
#include <sundials/sundials_core.h>    // Provides core SUNDIALS types
#include <arkode/arkode_arkstep.h>    // ARKStep provides explicit, implicit, IMEX additive RK methods.
#include <arkode/arkode_erkstep.h>    // ERKStep provides explicit RK methods.
#include <arkode/arkode_forcingstep.h> // ForcingStep provides a forcing method.
#include <arkode/arkode_lsrkstep.h>    // LSRKStep provides low-storage RK methods.
#include <arkode/arkode_mristep.h>    // MRISStep provides multirate RK methods.
#include <arkode/arkode_splittingstep.h> // SplittingStep provides operator splitting methods.
#include <arkode/arkode_sprkstep.h>    // SPRKStep provides symplectic partitioned RK methods.
```

Each of these define several types and various constants, include function prototypes, and include the shared `arkode/arkode.h` and `arkode/arkode_ls.h` header files. No other header files are required to be included, but there are optional ones that can be included as necessary. Information on optional headers is given in the relevant documentation section.

The calling program must also include an *N\_Vector* implementation header file, of the form `nvector/nvector_*.h`. See §8 for the appropriate name.

If the user includes a non-trivial implicit component to their ODE system in ARKStep, or if the slow time scale for MRISStep should be treated implicitly, then each implicit stage will require a nonlinear solver for the resulting system of algebraic equations – the default for this is a modified or inexact Newton iteration, depending on the user's choice of linear solver. If choosing to set which nonlinear solver module, or when interacting with a *SUNNonlinearSolver* module directly, the calling program must also include a *SUNNonlinearSolver* header file, of the form `sunnonlinsol/sunnonlinsol_***.h` where `***` is the name of the nonlinear solver module (see §11 for more information).

If using a nonlinear solver that requires the solution of a linear system of the form  $Ax = b$  (e.g., the default Newton iteration), then a linear solver module header file will also be required. Similarly, if the ODE system in ARKStep involves a non-identity mass matrix  $M \neq I$ , then each time step will require a linear solver for systems of the form  $Mx = b$ . In this case it will be necessary to include the header file for a *SUNLinearSolver* solver, which is of the form `sunlinsol/sunlinsol_***.h` (see §10 for more information).

If the linear solver is matrix-based, the linear solver header will also include a header file of the form `sunmatrix/sunmatrix_*.h` where `*` is the name of the matrix implementation compatible with the linear solver (see §9 for more information).

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

needed to access the preconditioner initialization routines.

## 5.2 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP using ARKODE. Most of the steps are independent of the NVECTOR, SUNMATRIX, SUNLINSOL and SUNNONLINSOL implementations used. For the steps that are not, refer to §8, §9, §10, and §11 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. Create the SUNDIALS simulation context object.

Call `SUNContext_Create()` to allocate the `SUNContext` object.

3. 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`.

4. 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 `sunrealtype` 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);
```

For details on each of SUNDIALS' provided vector implementations, see the corresponding sections in §8 for details.

5. Create ARKODE object

Call a stepper-specific constructor, `arkode_mem = *StepCreate(...)`, to create the ARKODE memory block. These routines return a `void*` pointer to this memory structure. See §5.7.1.1, §5.8.1.1, §5.11.2.1, or §5.13.1.1 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. Alter-

natively, call `ARKodeWFTolerances()` to specify a function which sets directly the weights used in evaluating WRMS vector norms. See §5.3.2 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. Create matrix object

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

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

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

or similar for the other matrix modules (see §9 for further information).

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

#### 8. Create linear solver object

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

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

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

where `*` can be replaced with “Dense”, “SPGMR”, or other options, as discussed in §10.

#### 9. 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 §10 for details.

#### 10. Attach linear solver module

If a linear solver was created above for implicit stage solves, initialize the ARKLS linear solver interface by attaching the linear solver object (and Jacobian matrix object, if applicable) with the call (for details see §5.3.3):

```
ier = ARKodeSetLinearSolver(...);
```

Similarly, if the problem involves a non-identity mass matrix, initialize the ARKLS mass matrix linear solver interface by attaching the mass linear solver object (and mass matrix object, if applicable) with the call (for details see §5.3.3):

```
ier = ARKodeSetMassLinearSolver(...);
```

#### 11. Create nonlinear solver object

If the problem involves an implicit component, and if a non-default nonlinear solver object will be used for implicit stage solves (see §5.3.5), then the desired nonlinear solver object must be created by using the appropriate functions defined by the particular SUNNONLINSOL implementation (e.g., `NLS = SUNNonlinSol_***(...)`; where `***` is the name of the nonlinear solver (see §11 for details).

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

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

where `*` can be replaced with “Newton”, “FixedPoint”, or other options, as discussed in §11.

#### 12. Attach nonlinear solver module

If a nonlinear solver object was created above, then it must be attached to ARKODE using the call (for details see §5.3.5):

```
ier = ARKodeSetNonlinearSolver(...);
```

#### 13. Set nonlinear solver optional inputs

Call the appropriate set functions for the selected nonlinear solver module to change optional inputs specific to that nonlinear solver. These *must* be called after attaching the nonlinear solver to ARKODE, otherwise the optional inputs will be overridden by ARKODE defaults. See §11 for more information on optional inputs.

#### 14. Set optional inputs

Call `ARKodeSet*` functions to change any optional inputs that control the behavior of ARKODE from their default values. See §5.3.8 for details.

Additionally, call `*StepSet*` routines to change any stepper-specific optional inputs from their default values. See §5.7.1.8, §5.8.1.5, §5.11.2.7, or §5.13.1.4 for details.

#### 15. Specify rootfinding problem

Optionally, call `ARKodeRootInit()` to initialize a rootfinding problem to be solved during the integration of the ODE system. See §5.3.6 for general details, and §5.3.8 for relevant optional input calls.

#### 16. Advance solution in time

For each point at which output is desired, call

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

Here, `itask` specifies the return mode. The vector `yout` (which can be the same as the vector `y0` above) will contain  $y(t_{\text{out}})$ . See §5.3.7 for details.

#### 17. Get optional outputs

Call `ARKodeGet*` functions to obtain optional output. See §5.3.10 for details.

Additionally, call `*StepGet*` routines to retrieve any stepper-specific optional outputs. See §5.7.1.10, §5.8.1.7, §5.11.2.9, or §5.13.1.6 for details.

#### 18. Deallocate memory for solution vector

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

```
N_VDestroy(y);
```

#### 19. Free solver memory

Call `ARKodeFree()` to free the memory allocated for the ARKODE module (and any nonlinear solver module).

#### 20. 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.

## 21. Free nonlinear solver memory

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

## 22. Free the SUNContext object

Call `SUNContext_Free()` to free the memory allocated for the `SUNContext` object.

## 23. Finalize MPI, if used

Call `MPI_Finalize` to terminate MPI.

## 5.3 ARKODE User-callable functions

This section describes the shared ARKODE functions that are called by the user to setup and then solve an IVP. Some of these are required; however, starting with §5.3.8, 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, §5.2, for the correct order of these calls.

On an error, each user-callable function returns a negative value (or NULL if the function returns a pointer) and sends an error message to the error handler, which prints the message to `stderr` by default. However, the user can set a file as error output or can provide their own error handler (see §4.3 for details).

We note that depending on the choice of time-stepping module, only a subset of ARKODE's user-callable functions will be applicable/supported. We thus categorize the functions below into five groups:

- A. functions that apply for all time-stepping modules,
- B. functions that apply for time-stepping modules that allow temporal adaptivity,
- C. functions that apply for time-stepping modules that utilize implicit solvers (nonlinear or linear),
- D. functions that apply for time-stepping modules that support non-identity mass matrices, and
- E. functions that apply for time-stepping modules that support relaxation Runge–Kutta methods.

In the function descriptions below, we identify those that have any of the restrictions B-E above. Then in the introduction for each of the stepper-specific documentation sections (§5.7.1, §5.8.1, §5.9.1, §5.10.1, §5.11.2, §5.12.2, and §5.13.1) we clarify the categories of these functions that are supported.

### 5.3.1 ARKODE initialization and deallocation functions

For functions to create an ARKODE stepper instance see `ARKStepCreate()`, `ERKStepCreate()`, `ForcingStepCreate()`, `LSRKStepCreateSTS()`, `LSRKStepCreateSSP()`, `MRISStepCreate()`, `SplittingStepCreate()`, or `SPRKStepCreate()`.

void **ARKodeFree**(void \*\*arkode\_mem)

This function frees the problem memory created a stepper constructor.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE stepper memory block.

**Returns**

none

Added in version 6.1.0: This function replaces stepper specific versions in `ARKStep`, `ERKStep`, `MRISStep`, and `SPRKStep`.

### 5.3.2 ARKODE tolerance specification functions

These functions specify the integration tolerances. One of them **should** be called before the first call to [ARKodeEvolve\(\)](#); 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  $v$ :

$$\|v\|_{W RMS} = \left( \frac{1}{N} \sum_{i=1}^N (v_i \text{ewt}_i)^2 \right)^{1/2},$$

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, [sunrealtype](#) reltol, [sunrealtype](#) abstol)

This function specifies scalar relative and absolute tolerances.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **reltol** – scalar relative tolerance.
- **abstol** – scalar absolute tolerance.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NO\_MALLOC** – `arkode_mem` was not allocated.
- **ARK\_ILL\_INPUT** – an argument had an illegal value (e.g. a negative tolerance).

#### Note

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.scalar\_tolerances”.

Added in version 6.1.0.

int **ARKodeSVtolerances**(void \*arkode\_mem, [sunrealtype](#) 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).

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **reltol** – scalar relative tolerance.



- **abstol** – vector containing the absolute tolerances for each solution component.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NO\_MALLOC** – `arkode_mem` was not allocated.
- **ARK\_ILL\_INPUT** – an argument had an illegal value (e.g. a negative tolerance).

Added in version 6.1.0.

int **ARKodeWftolerances**(void \*arkode\_mem, [ARKEwtFn](#) efun)

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

#### Parameters

- **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 values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NO\_MALLOC** – `arkode_mem` was not allocated.

Added in version 6.1.0.

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  $v$ :

$$\|v\|_{W RMS} = \left( \frac{1}{N} \sum_{i=1}^N (v_i \text{rwt}_i)^2 \right)^{1/2},$$

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, [sunrealtype](#) rabstol)

This function specifies a scalar absolute residual tolerance.

#### Parameters



- **arkode\_mem** – pointer to the ARKODE memory block.
- **rabstol** – scalar absolute residual tolerance.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NO\_MALLOC** – `arkode_mem` was not allocated.
- **ARK\_ILL\_INPUT** – an argument had an illegal value (e.g. a negative tolerance).

Added in version 6.1.0.

int **ARKodeResVtolerance**(void \*arkode\_mem, *N\_Vector* rabstol)

This function specifies a vector of absolute residual tolerances.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **rabstol** – vector containing the absolute residual tolerances for each solution component.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NO\_MALLOC** – `arkode_mem` was not allocated.
- **ARK\_ILL\_INPUT** – an argument had an illegal value (e.g. a negative tolerance).

Added in version 6.1.0.

int **ARKodeResFtolerance**(void \*arkode\_mem, *ARKRwtFn* rfun)

This function specifies a user-supplied function *rfun* to compute the residual weight vector *rwt*.

**Parameters**

- **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 values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NO\_MALLOC** – `arkode_mem` was not allocated.

Added in version 6.1.0.

### 5.3.2.1 General advice on the choice of tolerances

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

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

- (2) The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector  $y$  may be so small that pure relative error control is meaningless. For example, if  $y_i$  starts at some nonzero value, but in time decays to zero, then pure relative error control on  $y_i$  makes no sense (and is overly costly) after  $y_i$  is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. For example, see the example problem `ark_robertson.c`, and the discussion of it in the ARKODE Examples Documentation [87]. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the `atols` vector therein. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.
- (3) The residual absolute tolerances `rabstol` (whether scalar or vector) follow a similar explanation as for `abstol`, except that these should be set to the noise level of the equation components, i.e. the noise level of  $My$ . For problems in which  $M = I$ , it is recommended that `rabstol` be left unset, which will default to the already-supplied `abstol` values.
- (4) Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual step. The final (global) errors are an accumulation of those per-step errors, where that accumulation factor is problem-dependent. A general rule of thumb is to reduce the tolerances by a factor of 10 from the actual desired limits on errors. So if you want .01% relative accuracy (globally), a good choice for `reltol` is  $10^{-5}$ . In any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

### 5.3.2.2 Advice on controlling nonphysical negative values

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

- (1) The best way to control the size of unwanted negative computed values is with tighter absolute tolerances. Again this requires some knowledge of the noise level of these components, which may or may not be different for different components. Some experimentation may be needed.
- (2) If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in  $y$  returned by 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) Some of ARKODE's time stepping modules support component-wise constraints on solution components,  $y_i < 0$ ,  $y_i \leq 0$ ,  $y_i > 0$ , or  $y_i \geq 0$ , through the user-callable function `ARKodeSetConstraints()`. At each internal time step, if any constraint is violated then ARKODE will attempt a smaller time step that should not violate this constraint. This reduced step size is chosen such that the step size is the largest possible but where the solution component satisfies the constraint.
- (5) For time-stepping modules that support temporal adaptivity, positivity and non-negativity constraints on components can also be enforced by use of the recoverable error return feature in the user-supplied right-hand side function(s). When a recoverable error is encountered, ARKODE will retry the step with a smaller step size, which typically alleviates the problem. However, since this reduced step size is chosen without knowledge of the

solution constraint, it may be overly conservative. Thus this option involves some additional overhead cost, and should only be exercised if the above recommendations are unsuccessful.

### 5.3.3 Linear solver interface functions

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

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

where

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

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

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

Matrix-free `SUNLinearSolver` modules instead use iterative methods to solve the Newton systems of equations, and only require the *action* of the matrix on a vector,  $\mathcal{A}v$ . With most of these methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver portions of §5.3.8 and §5.4.

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 `*StepCreate` but before any calls to `ARKodeEvolve()`, the user's program must create the appropriate `SUNLinearSolver` object and call the function `ARKodeSetLinearSolver()`, as documented below. To create the `SUNLinearSolver` object, the user may call one of the SUNDIALS-packaged `SUNLinSol` module constructor routines via a call of the form

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

The current list of SUNDIALS-packaged `SUNLinSol` modules, and their constructor routines, may be found in chapter §10. Alternately, a user-supplied `SUNLinearSolver` module may be created and used. Specific information on how to create such user-provided modules may be found in §10.1.8.

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

int **ARKodeSetLinearSolver**(void \*arkode\_mem, *SUNLinearSolver* LS, *SUNMatrix* J)

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

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **LS** – the `SUNLinearSolver` object to use.

- **J** – the template Jacobian `SUNMatrix` object to use (or `NULL` if not applicable).

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was `NULL`.
- **ARKLS\_MEM\_FAIL** – there was a memory allocation failure.
- **ARKLS\_ILL\_INPUT** – ARKLS is incompatible with the provided *LS* or *J* input objects, or the current `N_Vector` module.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

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

If *LS* is a matrix-based linear solver, then the template Jacobian matrix *J* will be used in the solve process, so if additional storage is required within the `SUNMatrix` object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular `SUNMATRIX` type in the §9 for further information).

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

Added in version 6.1.0.

### 5.3.4 Mass matrix solver specification functions

As discussed in §2.15.6, if the ODE system involves a non-identity mass matrix  $M \neq I$ , then ARKODE must solve linear systems of the form

$$Mx = b.$$

ARKODE's ARKLS mass-matrix linear solver interface supports all valid `SUNLinearSolver` modules for this task. For iterative linear solvers, user-supplied preconditioning can be applied. For the specification of a preconditioner, see the iterative linear solver portions of §5.3.8 and §5.4. If preconditioning is to be performed, user-supplied functions should be used to define left and right preconditioner matrices  $P_1$  and  $P_2$  (either of which could be the identity matrix), such that the product  $P_1 P_2$  approximates the mass matrix *M*.

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

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

- If both are matrix-based, then they must utilize the same `SUNMatrix` type, since these will be added when forming the Newton system matrix  $\mathcal{A}$ . In this case, both the Newton and mass matrix linear solver interfaces can use the same `SUNLinearSolver` object, although different solver objects (e.g. with different solver parameters) are also allowed.
- If both are matrix-free, then the Newton and mass matrix `SUNLinearSolver` objects must be different. These may even use different solver algorithms (SPGMR, SPBCGS, etc.), if desired. For example, if the mass matrix is symmetric but the Jacobian is not, then PCG may be used for the mass matrix systems and SPGMR for the Newton systems.

```
int ARKodeSetMassLinearSolver(void *arkode_mem, SUNLinearSolver LS, SUNMatrix M, sunbooleantype
                             time_dep)
```

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

#### Parameters

- **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 ( $M \neq M(t)$ ). `SUNTRUE` indicates time-dependence of the mass matrix.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was `NULL`.
- **ARKLS\_MEM\_FAIL** – there was a memory allocation failure.
- **ARKLS\_ILL\_INPUT** – ARKLS is incompatible with the provided *LS* or *M* input objects, or the current `N_Vector` module.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

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

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

If called with *time\_dep* set to `SUNFALSE`, then the mass matrix is only computed and factored once (or when either `*StepReInit` or `ARKodeResize()` are called), with the results reused throughout the entire ARKODE simulation.

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

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

Added in version 6.1.0.

### 5.3.5 Nonlinear solver interface functions

When changing the nonlinear solver in ARKODE, after the call to `*StepCreate` but before any calls to [ARKodeEvolve\(\)](#), the user's program must create the appropriate `SUNNonlinearSolver` object and call [ARKodeSetNonlinearSolver\(\)](#), as documented below. If any calls to [ARKodeEvolve\(\)](#) have been made, then ARKODE will need to be reinitialized by calling `*StepReInit` to ensure that the nonlinear solver is initialized correctly before any subsequent calls to [ARKodeEvolve\(\)](#).

The first argument passed to the routine [ARKodeSetNonlinearSolver\(\)](#) is the ARKODE memory pointer returned by `*StepCreate`; the second argument passed to this function is the desired `SUNNonlinearSolver` object to use for solving the nonlinear system for each implicit stage. A call to this function attaches the nonlinear solver to the main ARKODE integrator.

int [ARKodeSetNonlinearSolver](#)(void \*arkode\_mem, [SUNNonlinearSolver](#) NLS)

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

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **NLS** – the `SUNNonlinearSolver` object to use.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_MEM\_FAIL** – there was a memory allocation failure.
- **ARK\_ILL\_INPUT** – ARKODE is incompatible with the provided *NLS* input object.
- **ARK\_STEPPER\_UNSUPPORTED** – nonlinear solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

ARKODE will use the Newton `SUNNonlinearSolver` module by default; a call to this routine replaces that module with the supplied *NLS* object.

Added in version 6.1.0.

### 5.3.6 Rootfinding initialization function

As described in §2.16, while solving the IVP, ARKODE's time-stepping modules have the capability to find the roots of a set of user-defined functions. To activate the root-finding algorithm, call the following function. This is normally called only once, prior to the first call to [ARKodeEvolve\(\)](#), but if the rootfinding problem is to be changed during the solution, [ARKodeRootInit\(\)](#) can also be called prior to a continuation call to [ARKodeEvolve\(\)](#).

**Note**

The solution is interpolated to the times at which roots are found.

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 \*StepCreate, and before *ARKodeEvolve()*.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nrtfn** – number of functions  $g_i$ , an integer  $\geq 0$ .
- **g** – name of user-supplied function, of type *ARKRootFn()*, defining the functions  $g_i$  whose roots are sought.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_MEM\_FAIL** – there was a memory allocation failure.
- **ARK\_ILL\_INPUT** – *nrtfn* is greater than zero but *g* is NULL.

**Note**

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 \*StepReInit, where the new IVP has no rootfinding problem but the prior one did, then call *ARKodeRootInit* with *nrtfn* = 0.

Added in version 6.1.0.

### 5.3.7 ARKODE solver function

This is the central step in the solution process – the call to perform the integration of the IVP. The input argument *itask* specifies one of two modes as to where 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 **ARKodeEvolve**(void \*arkode\_mem, *sunrealtype* tout, *N\_Vector* yout, *sunrealtype* \*tret, int itask)

Integrates the ODE over an interval in  $t$ .

**Parameters**

- **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 just overtaken a user-specified output time, *tout*, in the direction of integration, i.e.  $t_{n-1} < tout \leq t_n$  for



forward integration, or  $t_n \leq tout < t_{n-1}$  for backward integration. If interpolation is enabled (on by default), it will then compute an approximation to the solution  $y(tout)$  by interpolation (as described in §2.2). Otherwise, the solution at the time reached by the solver is returned,  $y(tret)$ .

The `ARK_ONE_STEP` option tells the solver to only take a single internal step,  $y_{n-1} \rightarrow y_n$ , and return the solution at that point,  $y_n$ , in the vector *yout*.

### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_ROOT\_RETURN** – `ARKodeEvolve()` succeeded, and found one or more roots. If the number of root functions, *nrtfn*, is greater than 1, call `ARKodeGetRootInfo()` to see which  $g_i$  were found to have a root at (*\*tret*).
- **ARK\_TSTOP\_RETURN** – `ARKodeEvolve()` succeeded and returned at *tstop*.
- **ARK\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARK\_NO\_MALLOC** – *arkode\_mem* was not allocated.
- **ARK\_ILL\_INPUT** – one of the inputs to `ARKodeEvolve()` 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:
  - (a) A component of the error weight vector became zero during internal time-stepping.
  - (b) The linear solver initialization function (called by the user after calling *\*StepCreate*) failed to set the linear solver-specific *lsolve* field in *arkode\_mem*.
  - (c) A root of one of the root functions was found both at a point *t* and also very near *t*.
  - (d) The initial condition violates the inequality constraints.
- **ARK\_TOO\_MUCH\_WORK** – the solver took *mxstep* internal steps but could not reach *tout*. The default value for *mxstep* is `MXSTEP_DEFAULT = 500`.
- **ARK\_TOO\_MUCH\_ACC** – the solver could not satisfy the accuracy demanded by the user for some internal step.
- **ARK\_ERR\_FAILURE** – error test failures occurred either too many times (*ark\_maxnef*) during one internal time step or occurred with  $|h| = h_{min}$ .
- **ARK\_CONV\_FAILURE** – either convergence test failures occurred too many times (*ark\_maxncf*) during one internal time step or occurred with  $|h| = h_{min}$ .
- **ARK\_LINIT\_FAIL** – the linear solver’s initialization function failed.
- **ARK\_LSETUP\_FAIL** – the linear solver’s setup routine failed in an unrecoverable manner.
- **ARK\_LSOLVE\_FAIL** – the linear solver’s solve routine failed in an unrecoverable manner.
- **ARK\_MASSINIT\_FAIL** – the mass matrix solver’s initialization function failed.
- **ARK\_MASSSETUP\_FAIL** – the mass matrix solver’s setup routine failed.
- **ARK\_MASSSOLVE\_FAIL** – the mass matrix solver’s solve routine failed.
- **ARK\_VECTOROP\_ERR** – a vector operation error occurred.
- **ARK\_DOMEIG\_FAIL** – the dominant eigenvalue function failed. It is either not provided or returns an illegal value.
- **ARK\_MAX\_STAGE\_LIMIT\_FAIL** – stepper failed to achieve stable results. Either reduce the step size or increase the *stage\_max\_limit*



**Note**

The input vector *yout* can use the same memory as the vector *y0* of initial conditions that was passed to *\*StepCreate*.

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 *ARKodeEvolve()* failures.

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

On any error return in which one or more internal steps were taken by *ARKodeEvolve()*, 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.

Added in version 6.1.0.

### 5.3.8 Optional input functions

There are numerous optional input parameters that control the behavior of ARKODE, 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 ARKODE options (*Optional inputs for ARKODE*),
- Step adaptivity solver options (*Optional inputs for time step adaptivity*),
- Implicit stage solver options (*Optional inputs for implicit stage solves*),
- Linear solver interface options (*Linear solver interface optional input functions*), and
- Rootfinding options (*Rootfinding optional input functions*).

For the most casual use of ARKODE, relying on the default set of solver parameters, the reader can skip to section on user-supplied functions, §5.4.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. All error return values are negative, so a test on the return arguments for negative values will catch all errors. Finally, a call to an *ARKodeSet\*\*\** function can generally be made from the user's calling program at any time *after* creation of the ARKODE solver via *\*StepCreate*, and, the function exited successfully, takes effect immediately. *ARKodeSet\*\*\** functions that cannot be called at any time note this in the "notes" section of the function documentation.

## 5.3.8.1 Optional inputs for ARKODE

Optional input	Function name	Default
Set ARKODE options from the command line or file	<i>ARKodeSetOptions()</i>	internal
Return ARKODE parameters to their defaults	<i>ARKodeSetDefaults()</i>	internal
Set integrator method order	<i>ARKodeSetOrder()</i>	stepper-specific
Set dense output interpolation type	<i>ARKodeSetInterpolantType()</i>	stepper-specific
Set dense output polynomial degree	<i>ARKodeSetInterpolantDegree()</i>	method-dependent
Disable time step adaptivity (fixed-step mode)	<i>ARKodeSetFixedStep()</i>	disabled
Set forward or backward integration direction	<i>ARKodeSetStepDirection()</i>	0.0
Supply an initial step size to attempt	<i>ARKodeSetInitStep()</i>	estimated
Maximum no. of warnings for $t_n + h = t_n$	<i>ARKodeSetMaxHnilWarns()</i>	10
Maximum no. of internal steps before <i>tout</i>	<i>ARKodeSetMaxNumSteps()</i>	500
Maximum absolute step size	<i>ARKodeSetMaxStep()</i>	$\infty$
Minimum absolute step size	<i>ARKodeSetMinStep()</i>	0.0
Set a value for $t_{stop}$	<i>ARKodeSetStopTime()</i>	undefined
Interpolate at $t_{stop}$	<i>ARKodeSetInterpolateStopTime()</i>	SUNFALSE
Disable the stop time	<i>ARKodeClearStopTime()</i>	N/A
Supply a pointer for user data	<i>ARKodeSetUserData()</i>	NULL
Maximum no. of ARKODE error test failures	<i>ARKodeSetMaxErrTestFails()</i>	7
Set inequality constraints on solution	<i>ARKodeSetConstraints()</i>	NULL
Set max number of constraint failures	<i>ARKodeSetMaxNumConstrFails()</i>	10
Set the checkpointing scheme to use (for adjoint)	<i>ARKodeSetAdjointCheckpointScheme()</i>	NULL
Set the checkpointing step index (for adjoint)	<i>ARKodeSetAdjointCheckpointIndex()</i>	0
Use compensated summation for accumulating time	<i>ARKodeSetUseCompensatedSums()</i>	SUNFALSE

int **ARKodeSetOptions**(void \*arkode\_mem, const char \*arkid, const char \*file\_name, int argc, char \*argv[])

Sets ARKODE options from an array of strings or a file.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **arkid** – the prefix for options to read. The default is “arkode”.
- **file\_name** – the name of a file containing options to read. If this is NULL or an empty string, “”, then no file is read.
- **argc** – length of the argv array.
- **argv** – an array of strings containing the options to set and their values.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **other** – error return value from relevant ARKODE “set” routine.

Example usage:

In a C or C++ program, the following will enable command-line processing:

```
/* Create ARKODE memory block */
void* arkode_mem = ARKStepCreate(fe, fi, T0, y, ctx);

/* Configure ARKODE as normal */
...

/* Override settings with command-line options using default "arkode" prefix */
flag = ARKodeSetOptions(arkode_mem, NULL, NULL, argc, argv);
```

Then when running the program, the user can specify desired options, e.g.,

```
$ ./a.out arkode.order 3 arkode.interpolant_type ARK_INTERP_LAGRANGE
```

#### Note

The `argc` and `argv` arguments are typically those supplied to the user's main routine however, this is not required. The inputs are left unchanged by `ARKodeSetOptions()`.

If the `arkid` argument is `NULL`, then the default prefix, `arkode`, must be used for all ARKODE options. Whether `arkid` is supplied or not, a "." must be used to separate an option key from the prefix. For example, when using the default `arkid`, the option `arkode.order` followed by the value can be used to set the method order of accuracy.

When using a combination of ARKODE integrators (e.g., via `MRISplit`, `SplittingStep`, or `ForcingStep`), it is recommended that users call `ARKodeSetOptions()` for each ARKODE integrator using a distinct `arkid` so they can be controlled separately. For example, "fast" and "slow" option prefixes can be used to differentiate between options for the slow and fast integrators in an MRI method (i.e., `fast.order` and `slow.order` followed by the desired values to set the method order for the fast and slow time scales, respectively).

ARKODE options set via `ARKodeSetOptions()` will overwrite any previously set values. Options are set in the order they are given in `argv` and, if an option with the same prefix appears multiple times in `argv`, the value of the last occurrence will be used.

The supported option names are noted within the documentation for the corresponding ARKODE "set" function. For options that take a `sunboolean_t` as input, use 1 to indicate true and 0 for false.

#### Warning

This function is not available in the Fortran interface.

File-based options are not yet supported, so the `file_name` argument should be set to either `NULL` or the empty string "".

Added in version 6.5.0.

int **ARKodeSetDefaults**(void \*arkode\_mem)

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

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

**Note**

Does not change the *user\_data* pointer or any parameters within the specified time-stepping module.

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

Added in version 6.1.0.

int **ARKodeSetOrder**(void \*arkode\_mem, int ord)

Specifies the order of accuracy for the IVP integration method.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ord** – requested order of accuracy.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – this option is not supported by the time-stepping module.

**Note**

The range of supported orders and the default order are stepper-specific:

- For explicit methods,  $1 \leq ord \leq 9$ , and the default value is 4.
- For implicit methods,  $1 \leq ord \leq 5$ , and the default value is 4.
- For ImEx methods,  $2 \leq ord \leq 5$ , and the default value is 4.
- ForcingStep does not support this function.
- LSRKStep does not support this function. Use [LSRKStepSetSTSMMethod\(\)](#) and [LSRKStepSetSSPMethod\(\)](#) instead.
- For MRISep,  $1 \leq ord \leq 5$ , and the default value is 3.
- For SplittingStep,  $1 \leq ord$ , and the default value is 1.
- For SPRKStep,  $ord \in \{1, 2, 3, 4, 5, 6, 8, 10\}$ , and the default value is 4.

Since *ord* affects the memory requirements for the internal ARKODE memory block, it cannot be changed after the first call to [ARKodeEvolve\(\)](#), unless *\*StepReInit* is called.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.order”.

Added in version 6.1.0.

int **ARKodeSetInterpolantType**(void \*arkode\_mem, int itype)

Specifies the interpolation type used for dense output (interpolation of solution output values) and implicit method predictors. By default, Hermite interpolation is used except with SPRK methods where Lagrange interpolation is the default.

This routine must be called *after* the calling a stepper constructor. After the first call to [ARKodeEvolve\(\)](#) the interpolation type may not be changed without first calling a stepper `ReInit` function.

The Hermite interpolation module (ARK\_INTERP\_HERMITE) is described in §2.2.1, and the Lagrange interpolation module (ARK\_INTERP\_LAGRANGE) is described in §2.2.2. ARK\_INTERP\_NONE will disable interpolation.

When interpolation is disabled, using rootfinding is not supported, implicit methods must use the trivial predictor (the default option), and interpolation at stop times cannot be used (interpolating at stop times is disabled by default). With interpolation disabled, calling [ARKodeEvolve\(\)](#) in ARK\_NORMAL mode will return at or past the requested output time (setting a stop time may still be used to halt the integrator at a specific time).

Disabling interpolation will reduce the memory footprint of an integrator by two or more state vectors (depending on the interpolant type and degree) which can be beneficial when interpolation is not needed e.g., when integrating to a final time without output in between or using a solver from ARKODE as a fast time scale integrator with MRI methods.

This routine frees any previously-allocated interpolation module, and re-creates one according to the specified argument.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **itype** – requested interpolant type: ARK\_INTERP\_HERMITE, ARK\_INTERP\_LAGRANGE, or ARK\_INTERP\_NONE

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_MEM\_FAIL** – the interpolation module could not be allocated.
- **ARK\_ILL\_INPUT** – the *itype* argument is not recognized or the interpolation module has already been initialized.

#### Note

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.interpolant\_type”.

Changed in version 6.1.0: This function replaces stepper specific versions in ARKStep, ERKStep, MRIStep, and SPRKStep.

Added the ARK\_INTERP\_NONE option to disable interpolation.

Values set by a previous call to [ARKStepSetInterpolantDegree\(\)](#) are no longer nullified by a call to [ARKStepSetInterpolantType\(\)](#).

int **ARKodeSetInterpolantDegree**(void \*arkode\_mem, int degree)

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

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.

- **degree** – requested polynomial degree.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` or the interpolation module are NULL.
- **ARK\_INTERP\_FAIL** – this was called after `ARKodeEvolve()`.
- **ARK\_ILL\_INPUT** – an argument had an illegal value or the interpolation module has already been initialized.

#### Note

Allowed values are between 0 and 5.

This routine should be called *before* `ARKodeEvolve()`. After the first call to `ARKodeEvolve()` the interpolation degree may not be changed without first calling `*StepReInit`.

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

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

When  $q = 1$ , a linear interpolant is the default to ensure values obtained by the integrator are returned at the ends of the time interval.

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.interpolant\_degree”.

Added in version 6.1.0.

int **ARKodeSetFixedStep**(void \*arkode\_mem, *sunrealtype* hfixed)

Disables time step adaptivity within ARKODE, and specifies the fixed time step size to use for the following internal step(s).

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **hfixed** – value of the fixed step size to use.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

#### Note

Pass 0.0 to return ARKODE to the default (adaptive-step) mode – this is only allowed when using a time-stepping module that supports temporal adaptivity.

Use of this function is not generally recommended, since it gives 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()`, `ARKodeSetMaxErrTestFails()`, `ARKodeSetCFLFraction()`, `ARKodeSetErrorBias()`, `ARKodeSetFixedStepBounds()`, `ARKodeSetMaxCFailGrowth()`, `ARKodeSetMaxEFailGrowth()`, `ARKodeSetMaxFirstGrowth()`, `ARKodeSetMaxGrowth()`, `ARKodeSetMinReduction()`, `ARKodeSetSafetyFactor()`, `ARKodeSetSmallNumEFails()`, `ARKodeSetStabilityFn()`, `ARKodeSetAdaptController()`, and `ARKodeSetAdaptControllerByName()` 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 `ARKodeEvolve()` to resume integration.

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

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.fixed\_step”.

Added in version 6.1.0.

int **ARKodeSetStepDirection**(void \*arkode\_mem, *sunrealtype* stepdir)

Specifies the direction of integration (forward or backward).

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **stepdir** – value whose sign determines the direction. A positive value selects forward integration, a negative value selects backward integration, and zero leaves the current direction unchanged.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

#### Note

The step direction can only be set after a call to either `*Create`, `*StepReInit`, or `ARKodeReset()` but before a call to `ARKodeEvolve()`.

When the direction changes for an adaptive method, the adaptivity controller and next step size are reset. A new initial step size will be estimated at the next call to `ARKodeEvolve()` or can be specified with `ARKodeSetInitStep()`.

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.step\_direction”.

Added in version 6.2.0.

int **ARKodeSetInitStep**(void \*arkode\_mem, *sunrealtype* hin)

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

#### Parameters

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

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

**Note**

Pass 0.0 to use the default value – this is only allowed when using a time-stepping module that supports temporal adaptivity.

By default, ARKODE estimates the initial step size to be  $h = \sqrt{\frac{2}{\|\ddot{y}\|}}$ , where  $\ddot{y}$  is estimate of the second derivative of the solution at  $t_0$ .

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

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.init\_step”.

Added in version 6.1.0.

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.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **mxhnil** – maximum allowed number of warning messages ( $> 0$ ).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

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

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

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.max\_hnil\_warns”.

Added in version 6.1.0.

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.

**Parameters**



- **arkode\_mem** – pointer to the ARKODE memory block.
- **mxsteps** – maximum allowed number of internal steps.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

**Note**

Passing `mxsteps = 0` results in ARKODE using the default value (500).

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

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.max\_num\_steps”.

Added in version 6.1.0.

int **ARKodeSetMaxStep**(void \*arkode\_mem, *sunrealtype* hmax)

Specifies the upper bound on the magnitude of the time step size.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **hmax** – maximum absolute value of the time step size ( $\geq 0$ ).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

Pass `hmax ≤ 0.0` to set the default value of  $\infty$ .

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.max\_step”.

Added in version 6.1.0.

int **ARKodeSetMinStep**(void \*arkode\_mem, *sunrealtype* hmin)

Specifies the lower bound on the magnitude of the time step size.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **hmin** – minimum absolute value of the time step size ( $\geq 0$ ).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

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

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.min\_step”.

Added in version 6.1.0.

int **ARKodeSetStopTime**(void \*arkode\_mem, *sunrealtype* tstop)

Specifies the value of the independent variable  $t$  past which the solution is not to proceed.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **tstop** – stopping time for the integrator.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

**Note**

The default is that no stop time is imposed.

Once the integrator returns at a stop time, any future testing for `tstop` is disabled (and can be re-enabled only through a new call to `ARKodeSetStopTime()`).

A stop time not reached before a call to `*StepReInit` or `ARKodeReset()` will remain active but can be disabled by calling `ARKodeClearStopTime()`.

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.stop\_time”.

Added in version 6.1.0.

int **ARKodeSetInterpolateStopTime**(void \*arkode\_mem, *sunbooleantype* interp)

Specifies that the output solution should be interpolated when the current  $t$  equals the specified `tstop` (instead of merely copying the internal solution  $y_n$ ).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **interp** – flag indicating to use interpolation (1) or copy (0).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

**Note**

This routine will be called by `ARKodeSetOptions()` when using the key “`arkid.interpolate_stop_time`”.

Added in version 6.1.0.

int **ARKodeClearStopTime**(void \*arkode\_mem)

Disables the stop time set with `ARKodeSetStopTime()`.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

**Note**

The stop time can be re-enabled though a new call to `ARKodeSetStopTime()`.

This routine will be called by `ARKodeSetOptions()` when using the key “`arkid.clear_stop_time`”.

Added in version 6.1.0.

int **ARKodeSetUserData**(void \*arkode\_mem, void \*user\_data)

Specifies the user data block *user\_data* and attaches it to the main ARKODE memory block.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **user\_data** – pointer to the user data.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

**Note**

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 preconditioner functions, the call to this function must be made *before* any calls to `ARKodeSetLinearSolver()` and/or `ARKodeSetMassLinearSolver()`.

Added in version 6.1.0.

int **ARKodeSetMaxErrTestFails**(void \*arkode\_mem, int maxnef)

Specifies the maximum number of error test failures permitted in attempting one step, before returning with an error.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **maxnef** – maximum allowed number of error test failures ( $> 0$ ).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

The default value is 7; set  $\text{maxnef} \leq 0$  to specify this default.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.max\_err\_test\_fails”.

Added in version 6.1.0.

int **ARKodeSetConstraints**(void \*arkode\_mem, *N\_Vector* constraints)

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

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **constraints** – vector of constraint flags. Each component specifies the type of solution constraint:

$$\text{constraints}[i] = \begin{cases} 0.0 & \Rightarrow \text{no constraint is imposed on } y_i, \\ 1.0 & \Rightarrow y_i \geq 0, \\ -1.0 & \Rightarrow y_i \leq 0, \\ 2.0 & \Rightarrow y_i > 0, \\ -2.0 & \Rightarrow y_i < 0. \end{cases}$$

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – the constraints vector contains illegal values.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

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

After a call to `ARKodeResize()` inequality constraint checking will be disabled and a call to `ARKodeSetConstraints()` is required to re-enable constraint checking.

Since constraint-handling is performed through cutting time steps that would violate the constraints, it is possible that this feature will cause some problems to fail due to an inability to enforce constraints even at the minimum time step size. Additionally, the features `ARKodeSetConstraints()` and `ARKodeSetFixedStep()` are incompatible, and should not be used simultaneously.

Added in version 6.1.0.

int **ARKodeSetMaxNumConstrFails**(void \*arkode\_mem, int maxfails)

Specifies the maximum number of constraint failures in a step before ARKODE will return with an error.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **maxfails** – maximum allowed number of constrain failures.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

Passing `maxfails <= 0` results in ARKODE using the default value (10).

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.max\_num\_constr\_fails”.

Added in version 6.1.0.

int **ARKodeSetAdjointCheckpointScheme**(void \*arkode\_mem, *SUNAdjointCheckpointScheme* checkpoint\_scheme)

Specifies the *SUNAdjointCheckpointScheme* to use for saving states during the forward integration, and loading states during backward integration of an adjoint system.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **checkpoint\_scheme** – the checkpoint scheme to use, or NULL to disable checkpointing.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.

- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.3.0.

int **ARKodeSetAdjointCheckpointIndex**(void \*arkode\_mem, *suncountertype* step\_index)

Specifies the step index (that is step number) to insert the next checkpoint at.

This is incremented along with the step count, but it is useful to be able to reset this index during recomputations of missing states during the backward adjoint integration.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **step\_idx** – the step to insert the next checkpoint at.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.3.0.

int **ARKodeSetUseCompensatedSums**(void \*arkode\_mem, *sunboolean* onoff)

Specifies if compensated summations should be used within ARKODE where supported. Currently, all ARKODE modules support compensated summation for accumulating time.

SPRKStep also supports an alternative stepping algorithm based on compensated summation which will be enabled/disabled by this function. This increases the computational cost by 2 extra vector operations per stage and an additional 5 per time step. It also requires one extra vector to be stored. However, it is significantly more robust to roundoff error accumulation.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **onoff** – should compensated summation be used (1) or not (0)

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the ARKODE memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

#### Note

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.use\_compensated\_sums”.

### 5.3.8.2 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 §2.11.

Optional input	Function name	Default
Provide a <i>SUNAdaptController</i> for ARKODE to use	<i>ARKodeSetAdaptController()</i>	I
Specify a <i>SUNAdaptController</i> for ARKODE to use	<i>ARKodeSetAdaptControllerByName()</i>	I
Adjust the method order used in the controller	<i>ARKodeSetAdaptivityAdjustment()</i>	0
Explicit stability safety factor	<i>ARKodeSetCFLFraction()</i>	0.5
Time step error bias factor	<i>ARKodeSetErrorBias()</i>	1.0
Bounds determining no change in step size	<i>ARKodeSetFixedStepBounds()</i>	1.0 1.0
Maximum step growth factor on convergence fail	<i>ARKodeSetMaxCFailGrowth()</i>	0.25
Maximum step growth factor on error test fail	<i>ARKodeSetMaxEFailGrowth()</i>	0.3
Maximum first step growth factor	<i>ARKodeSetMaxFirstGrowth()</i>	10000.0
Maximum allowed general step growth factor	<i>ARKodeSetMaxGrowth()</i>	20.0
Minimum allowed step reduction factor on error test fail	<i>ARKodeSetMinReduction()</i>	0.1
Time step safety factor	<i>ARKodeSetSafetyFactor()</i>	0.9
Error fails before MaxEFailGrowth takes effect	<i>ARKodeSetSmallNumEFails()</i>	2
Explicit stability function	<i>ARKodeSetStabilityFn()</i>	none
Set accumulated error estimation type	<i>ARKodeSetAccumulatedErrorType()</i>	none
Reset accumulated error	<i>ARKodeResetAccumulatedError()</i>	

int **ARKodeSetAdaptController**(void \*arkode\_mem, *SUNAdaptController* C)

Sets a user-supplied time-step controller object.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **C** – user-supplied time adaptivity controller.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_MEM\_FAIL** – C was NULL and the I controller could not be allocated.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

If C is NULL then the I controller will be created (see §13.2).

This is only compatible with time-stepping modules that support temporal adaptivity.

Not all time-stepping modules are compatible with all types of *SUNAdaptController* objects. While all steppers that support temporal adaptivity support controllers with *SUNAdaptController\_Type* type **SUN\_ADAPTCONTROLLER\_H**, only MRISep supports inputs with type **SUN\_ADAPTCONTROLLER\_MRI\_H\_TOL**.

Added in version 6.1.0.

Changed in version 6.3.0: The default controller was changed from PID to I.

int **ARKodeSetAdaptControllerByName**(void \*arkode\_mem, const char \*cname)

Sets a user-supplied time step controller object by name.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **cname** – name of the time adaptivity controller to use. Allowable values currently include "Soderlind", "PID", "PI", "I", "ExpGus", "ImpGus", "ImExGus", "H0211", "H0321", "H211", and "H312". For information on these options, see §13.2 and §13.3.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_ILL\_INPUT** – cname did not match an allowed value.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

It is not possible to adjust the internal controller parameters when using this function. Users who wish to adjust these parameters should create and configure the [SUNAdaptController](#) object manually, and then call [ARKodeSetAdaptController\(\)](#).

Added in version 6.3.0.

int **ARKodeSetAdaptivityAdjustment**(void \*arkode\_mem, int adjust)

Called by a user to adjust the method order supplied to the temporal adaptivity controller. For example, if the user expects order reduction due to problem stiffness, they may request that the controller assume a reduced order of accuracy for the method by specifying a value *adjust* < 0.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **adjust** – adjustment factor (default is 0).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

This should be called prior to calling [ARKodeEvolve\(\)](#), and can only be reset following a call to [\\*StepReInit](#).

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.adaptivity\_adjustment”.

Added in version 6.1.0.

Changed in version 6.3.0: The default value was changed from -1 to 0



int **ARKodeSetCFLFraction**(void \*arkode\_mem, *sunrealtype* cfl\_frac)

Specifies the fraction of the estimated explicitly stable step to use.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **cfl\_frac** – maximum allowed fraction of explicitly stable step (default is 0.5).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.  
Any non-positive parameter will imply a reset to the default value.  
This routine will be called by *ARKodeSetOptions()* when using the key “arkid.cfl\_fraction”.

Added in version 6.1.0.

Changed in version 6.4.0: The restriction that cfl\_frac is less than one has been removed.

int **ARKodeSetErrorBias**(void \*arkode\_mem, *sunrealtype* bias)

Specifies the bias to be applied to the error estimates within accuracy-based adaptivity strategies.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **bias** – bias applied to error in accuracy-based time step estimation (default is 1.0).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.  
Any value below 1.0 will imply a reset to the default value.  
If both this and one of the stepper SetAdaptivityMethod functions or *ARKodeSetAdaptController()* will be called, then this routine must be called *second*.  
This routine will be called by *ARKodeSetOptions()* when using the key “arkid.error\_bias”.

Added in version 6.1.0.

Changed in version 6.3.0: The default value was changed from 1.5 to 1.0

int **ARKodeSetFixedStepBounds**(void \*arkode\_mem, *sunrealtype* lb, *sunrealtype* ub)

Specifies the step growth interval in which the step size will remain unchanged.

#### Parameters

- **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 1.0).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

Any interval *not* containing 1.0 will imply a reset to the default values.

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.fixed\_step\_bounds”.

Added in version 6.1.0.

Changed in version 6.3.0: The default upper bound was changed from 1.5 to 1.0

int **ARKodeSetMaxCFailGrowth**(void \*arkode\_mem, *sunrealtype* etacf)

Specifies the maximum step size growth factor upon an algebraic solver convergence failure on a stage solve within a step,  $\eta_{cf}$  from §2.15.3.1.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **etacf** – time step reduction factor on a nonlinear solver convergence failure (default is 0.25).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

Any value outside the interval  $(0, 1]$  will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.max\_cfail\_growth”.

Added in version 6.1.0.

int **ARKodeSetMaxEFailGrowth**(void \*arkode\_mem, *sunrealtype* etamxf)

Specifies the maximum step size growth factor upon multiple successive accuracy-based error failures in the solver.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **etamxf** – time step reduction factor on multiple error fails (default is 0.3).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

Any value outside the interval  $(0, 1]$  will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.max\_cfail\_growth”.

Added in version 6.1.0.

int **ARKodeSetMaxFirstGrowth**(void \*arkode\_mem, *sunrealtype* etamx1)

Specifies the maximum allowed growth factor in step size following the very first integration step.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **etamx1** – maximum allowed growth factor after the first time step (default is 10000.0).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

Any value  $\leq 1.0$  will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.max\_first\_growth”.

Added in version 6.1.0.

int **ARKodeSetMaxGrowth**(void \*arkode\_mem, *sunrealtype* mx\_growth)

Specifies the maximum allowed growth factor in step size between consecutive steps in the integration process.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **mx\_growth** – maximum allowed growth factor between consecutive time steps (default is 20.0).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

Any value  $\leq 1.0$  will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.max\_growth”.

Added in version 6.1.0.

int **ARKodeSetMinReduction**(void \*arkode\_mem, *sunrealtype* eta\_min)

Specifies the minimum allowed reduction factor in step size between step attempts, resulting from a temporal error failure in the integration process.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **eta\_min** – minimum allowed reduction factor in time step after an error test failure (default is 0.1).

#### Return values

- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.
- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

Any value outside the interval  $(0, 1)$  will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.min\_reduction”.

Added in version 6.1.0.

int **ARKodeSetSafetyFactor**(void \*arkode\_mem, *sunrealtype* safety)

Specifies the safety factor to be applied to the accuracy-based estimated step.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **safety** – safety factor applied to accuracy-based time step (default is 0.9).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support temporal adaptivity.

Any value  $\leq 0$  will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.safety\_factor”.

Added in version 6.1.0.

Changed in version 6.3.0: The default default was changed from 0.96 to 0.9. The maximum value is now exactly 1.0 rather than strictly less than 1.0.

int **ARKodeSetSmallNumEFails**(void \*arkode\_mem, int small\_nef)

Specifies the threshold for “multiple” successive error failures before the *etamxf* parameter from [ARKodeSetMaxEFailGrowth\(\)](#) is applied.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **small\_nef** – bound to determine ‘multiple’ for *etamxf* (default is 2).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

Any value  $\leq 0$  will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.small\_num\_efails”.

Added in version 6.1.0.

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.

**Parameters**

- **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 values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

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  $f^E(t, y)$  contains stiff terms.

Added in version 6.1.0.

The following routines are used to control algorithms that ARKODE can use to estimate the accumulated temporal error over multiple time steps. While these may be informational for users on their applications, this functionality is required when using multirate temporal adaptivity in MRISep via the [SUNAdaptController\\_MRIHTol](#) module. For time-stepping modules that compute both a solution and embedding,  $y_n$  and  $\tilde{y}_n$ , these may be combined to create a vector-valued local temporal error estimate for the current internal step,  $y_n - \tilde{y}_n$ . These local errors may be accumulated by ARKODE in a variety of ways, as determined by the enumerated type [ARKAccumError](#). In each of the cases below, the accumulation is taken over all steps since the most recent call to either [ARKodeSetAccumulatedErrorType\(\)](#) or [ARKodeResetAccumulatedError\(\)](#). Below the set  $S$  contains the indices of the steps since the last call to either of the aforementioned functions. The norm is taken using the tolerance-informed error-weight vector (see [ARKodeGetErrWeights\(\)](#)), and `reltol` is the user-specified relative solution tolerance.

enum **ARKAccumError**

The type of error accumulation that ARKODE should use.

Added in version 6.2.0.

enumerator **ARK\_ACCUMERROR\_NONE**

No accumulation should be performed

enumerator **ARK\_ACCUMERROR\_MAX**

Computes  $\text{reitol} \max_{i \in S} \|y_i - \tilde{y}_i\|_{WRMS}$

enumerator **ARK\_ACCUMERROR\_SUM**

Computes  $\text{reitol} \sum_{i \in S} \|y_i - \tilde{y}_i\|_{WRMS}$

enumerator **ARK\_ACCUMERROR\_AVG**

Computes  $\frac{\text{reitol}}{\Delta t_S} \sum_{i \in S} h_i \|y_i - \tilde{y}_i\|_{WRMS}$ , where  $h_i$  is the step size used when computing  $y_i$ , and  $\Delta t_S$  denotes the elapsed time over which  $S$  is taken.

int **ARKodeSetAccumulatedErrorType**(void \*arkode\_mem, *ARKAccumError* accum\_type)

Sets the strategy to use for accumulating a temporal error estimate over multiple time steps. By default, ARKODE will not accumulate any local error estimates (i.e., the default *accum\_type* is **ARK\_ACCUMERROR\_NONE**).

A non-default error accumulation strategy can be disabled by calling *ARKodeSetAccumulatedErrorType()* with the argument **ARK\_ACCUMERROR\_NONE**.

#### Note

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.accumulated\_error\_type”.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **accum\_type** – accumulation strategy.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_STEPPER\_UNSUPPORTED** – temporal error estimation is not supported by the current time-stepping module.

Added in version 6.2.0.

int **ARKodeResetAccumulatedError**(void \*arkode\_mem)

Resets the accumulated temporal error estimate, that was triggered by a previous call to *ARKodeSetAccumulatedErrorType()*.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_STEPPER\_UNSUPPORTED** – temporal error estimation is not supported by the current time-stepping module.

**Note**

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.reset\_accumulated\_error”.

Added in version 6.2.0.

**5.3.8.3 Optional inputs for implicit stage solves**

The mathematical explanation for the nonlinear solver strategies used by ARKODE, including how each of the parameters below is used within the code, is provided in §2.15.1.

Optional input	Function name	Default
Specify that the implicit RHS is linear	<code>ARKodeSetLinear()</code>	SUN- FALSE
Specify that the implicit RHS nonlinear	<code>ARKodeSetNonlinear()</code>	SUNTRUE
Specify that the implicit RHS is autonomous	<code>ARKodeSetAutonomous()</code>	SUN- FALSE
Implicit predictor method	<code>ARKodeSetPredictorMethod()</code>	0
User-provided implicit stage predictor	<code>ARKodeSetStagePredictFn()</code>	NULL
RHS function for nonlinear system evaluations	<code>ARKodeSetNlsRhsFn()</code>	NULL
Maximum number of nonlinear iterations	<code>ARKodeSetMaxNonlinIters()</code>	3
Coefficient in the nonlinear convergence test	<code>ARKodeSetNonlinConvCoef()</code>	0.1
Nonlinear convergence rate constant	<code>ARKodeSetNonlinCRDown()</code>	0.3
Nonlinear residual divergence ratio	<code>ARKodeSetNonlinRDiv()</code>	2.3
Maximum number of convergence failures	<code>ARKodeSetMaxConvFails()</code>	10
Specify if the implicit RHS is deduced after a nonlinear solve	<code>ARKodeSetDeduceImplicitRhs()</code>	SUN- FALSE

int **ARKodeSetLinear**(void \*arkode\_mem, int timedepend)

Specifies that the implicit portion of the problem is linear.

**Parameters**

- **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).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.



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 or inexact Newton iteration (not the fixed-point solver).

When  $f^I(t, y)$  is time-dependent, all linear solver structures (Jacobian, preconditioner) will be updated preceding *each* implicit stage. Thus one must balance the relative costs of such recomputation against the benefits of requiring only a single Newton linear solve.

Added in version 6.1.0.

int **ARKodeSetNonlinear**(void \*arkode\_mem)

Specifies that the implicit portion of the problem is nonlinear.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

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.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.nonlinear”.

Added in version 6.1.0.

int **ARKodeSetAutonomous**(void \*arkode\_mem, *sunbooleantype* autonomous)

Specifies that the implicit portion of the problem is autonomous i.e., does not explicitly depend on time.

When using an implicit or ImEx method with the trivial predictor, this option enables reusing the implicit right-hand side evaluation at the predicted state across stage solves within a step. This reuse reduces the total number of implicit RHS function evaluations.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **autonomous** – flag denoting if the implicit RHS function,  $f^I(t, y)$ , is autonomous (SUNTRUE) or non-autonomous (SUNFALSE).

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

### Warning

Results may differ when enabling both `ARKodeSetAutonomous()` and `ARKodeSetDeduceImplicitRhs()` with a stiffly accurate implicit method and using the trivial predictor. The differences are due to reusing the deduced implicit right-hand side (RHS) value in the initial nonlinear residual computation rather than evaluating the implicit RHS function. The significance of the difference will depend on how well the deduced RHS approximates the RHS evaluated at the trivial predictor. This behavior can be observed in `examples/arkode/C_serial/ark_brusselator.c` by comparing the outputs with `ARKodeSetAutonomous()` enabled/disabled.

Similarly programs that assume the nonlinear residual will always call the implicit RHS function will need to be updated to account for the RHS value reuse when using `ARKodeSetAutonomous()`. For example, `examples/arkode/C_serial/ark_KrylovDemo_prec.c` assumes that the nonlinear residual will be called and will evaluate the implicit RHS function before calling the preconditioner setup function. Based on this assumption, this example code saves some computations in the RHS evaluation for reuse in the preconditioner setup. However, when `ARKodeSetAutonomous()` is enabled, the call to the nonlinear residual before the preconditioner setup reuses a saved RHS evaluation and the saved data is actually from an earlier RHS evaluation that is not consistent with the state and RHS values passed to the preconditioner setup function. For this example, the code should not save data in the RHS evaluation but instead evaluate the necessary quantities within the preconditioner setup function using the input values.

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.autonomous”.

Added in version 6.1.0.

int **ARKodeSetPredictorMethod**(void \*arkode\_mem, int method)

Specifies the method from §2.15.5 to use for predicting implicit solutions.

### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **method** – method choice ( $0 \leq \text{method} \leq 4$ ):
  - 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,

### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The default value is 0. If *method* is set to an undefined value, this default predictor will be used.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.predictor\_method”.

Added in version 6.1.0.

int **ARKodeSetStagePredictFn**(void \*arkode\_mem, [ARKStagePredictFn](#) PredictStage)

Sets the user-supplied function to update the implicit stage predictor prior to execution of the nonlinear or linear solver algorithms that compute the implicit stage solution.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **PredictStage** – name of user-supplied predictor function. If NULL, then any previously-provided stage prediction function will be disabled.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

See §5.4.6 for more information on this user-supplied routine.

Added in version 6.1.0.

int **ARKodeSetNlsRhsFn**(void \*arkode\_mem, [ARKRhsFn](#) nls\_fn)

Specifies an alternative implicit right-hand side function for evaluating  $f^I(t, y)$  within nonlinear system function evaluations (2.39) - (2.41).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nls\_fn** – the alternative C function for computing the right-hand side function  $f^I(t, y)$  in the ODE.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The default is to use the implicit right-hand side function provided to the stepper constructor in nonlinear system functions. If the input implicit right-hand side function is `NULL`, the default is used.

When using a non-default nonlinear solver, this function must be called *after* `ARKodeSetNonlinearSolver()`.

Added in version 6.1.0.

int **ARKodeSetMaxNonlinIters**(void \*arkode\_mem, int maxcor)

Specifies the maximum number of nonlinear solver iterations permitted per implicit stage solve within each time step.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **maxcor** – maximum allowed solver iterations per stage ( $> 0$ ).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was `NULL`.
- **ARK\_ILL\_INPUT** – an argument had an illegal value or if the `SUNNONLINSOL` module is `NULL`.
- **ARK\_NLS\_OP\_ERR** – the `SUNNONLINSOL` object returned a failure flag.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The default value is 3; set `maxcor  $\leq$  0` to specify this default.

This routine will be called by `ARKodeSetOptions()` when using the key “`arkid.max_nonlin_iters`”.

Added in version 6.1.0.

int **ARKodeSetNonlinConvCoef**(void \*arkode\_mem, *sunrealtype* nlscoef)

Specifies the safety factor  $\epsilon$  used within the nonlinear solver convergence test (2.54).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nlscoef** – coefficient in nonlinear solver convergence test ( $> 0.0$ ).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was `NULL`.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The default value is 0.1; set *nlscoef*  $\leq 0$  to specify this default.

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.nonlin\_conv\_coef”.

Added in version 6.1.0.

int **ARKodeSetNonlinCRDown**(void \*arkode\_mem, *sunrealtype* crdown)

Specifies the constant  $c_r$  used in estimating the nonlinear solver convergence rate (2.53).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **crdown** – nonlinear convergence rate estimation constant (default is 0.3).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Any non-positive parameter will imply a reset to the default value.

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.nonlin\_crdown”.

Added in version 6.1.0.

int **ARKodeSetNonlinRDiv**(void \*arkode\_mem, *sunrealtype* rdiv)

Specifies the nonlinear correction threshold  $r_{div}$  from (2.55), beyond which the iteration will be declared divergent.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **rdiv** – tolerance on nonlinear correction size ratio to declare divergence (default is 2.3).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Any non-positive parameter will imply a reset to the default value.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.nonlin\_rdiv”.

Added in version 6.1.0.

int **ARKodeSetMaxConvFails**(void \*arkode\_mem, int maxncf)

Specifies the maximum number of nonlinear solver convergence failures permitted during one step,  $max_{ncf}$  from §2.15.3.1, before ARKODE will return with an error.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **maxncf** – maximum allowed nonlinear solver convergence failures per step ( $> 0$ ).

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The default value is 10; set  $max_{ncf} \leq 0$  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\(\)](#)).

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.max\_conv\_fails”.

Added in version 6.1.0.

int **ARKodeSetDeduceImplicitRhs**(void \*arkode\_mem, *sunbooleantype* deduce)

Specifies if implicit stage derivatives are deduced without evaluating  $f^I$ . See §2.15.1 for more details.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **deduce** – if SUNFALSE (default), the stage derivative is obtained by evaluating  $f^I$  with the stage solution returned from the nonlinear solver. If SUNTRUE, the stage derivative is deduced without an additional evaluation of  $f^I$ .

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.deduce\_implicit\_rhs”.

Added in version 6.1.0.

### 5.3.8.4 Linear solver interface optional input functions

The mathematical explanation of the linear solver methods available to ARKODE is provided in §2.15.2. We group the user-callable routines into four categories: general routines concerning the update frequency for matrices and/or preconditioners, optional inputs for matrix-based linear solvers, optional inputs for matrix-free linear solvers, and optional inputs for iterative linear solvers. We note that the matrix-based and matrix-free groups are mutually exclusive, whereas the “iterative” tag can apply to either case.

#### Optional inputs for the ARKLS linear solver interface

As discussed in §2.15.2.3, ARKODE strives to reuse matrix and preconditioner data for as many solves as possible to amortize the high costs of matrix construction and factorization. To that end, ARKODE provides user-callable routines to modify this behavior. Recall that the Newton system matrices that arise within an implicit stage solve are  $\mathcal{A}(t, z) \approx M(t) - \gamma J(t, z)$ , where the implicit right-hand side function has Jacobian matrix  $J(t, z) = \frac{\partial f^I(t, z)}{\partial z}$ .

The matrix or preconditioner for  $\mathcal{A}$  can only be updated within a call to the linear solver “setup” routine. In general, the frequency with which the linear solver setup routine is called may be controlled with the *msbp* argument to `ARKodeSetLSetupFrequency()`. When this occurs, the validity of  $\mathcal{A}$  for successive time steps intimately depends on whether the corresponding  $\gamma$  and  $J$  inputs remain valid.

At each call to the linear solver setup routine the decision to update  $\mathcal{A}$  with a new value of  $\gamma$ , and to reuse or reevaluate Jacobian information, depends on several factors including:

- the success or failure of previous solve attempts,
- the success or failure of the previous time step attempts,
- the change in  $\gamma$  from the value used when constructing  $\mathcal{A}$ , and
- the number of steps since Jacobian information was last evaluated.

Jacobian information is considered out-of-date when *msbj* or more steps have been completed since the last update, in which case it will be recomputed during the next linear solver setup call. The value of *msbj* is controlled with the *msbj* argument to `ARKodeSetJacEvalFrequency()`.

For linear-solvers with user-supplied preconditioning the above factors are used to determine whether to recommend updating the Jacobian information in the preconditioner (i.e., whether to set *jok* to `SUNFALSE` in calling the user-supplied `ARKLSPrecSetupFn`). For matrix-based linear solvers these factors determine whether the matrix  $J(t, y) = \frac{\partial f^I(t, y)}{\partial y}$  should be updated (either with an internal finite difference approximation or a call to the user-supplied `ARKLSJacFn`); if not then the previous value is reused and the system matrix  $\mathcal{A}(t, y) \approx M(t) - \gamma J(t, y)$  is recomputed using the current  $\gamma$  value.

Table 5.1: Optional inputs for the ARKLS linear solver interface

Optional input	Function name	Default
Max change in step signaling new $J$	<a href="#"><code>ARKodeSetDeltaGammaMax()</code></a>	0.2
Linear solver setup frequency	<a href="#"><code>ARKodeSetLSetupFrequency()</code></a>	20
Jacobian / preconditioner update frequency	<a href="#"><code>ARKodeSetJacEvalFrequency()</code></a>	51

int **ARKodeSetDeltaGammaMax**(void \*arkode\_mem, *sunrealtype* dgmax)

Specifies a scaled step size ratio tolerance,  $\Delta\gamma_{max}$  from §2.15.2.3, beyond which the linear solver setup routine will be signaled.

#### Parameters

- **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 values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Any non-positive parameter will imply a reset to the default value.

This routine will be called by [`ARKodeSetOptions\(\)`](#) when using the key “arkid.delta\_gamma\_max”.

Added in version 6.1.0.

int **ARKodeSetLSetupFrequency**(void \*arkode\_mem, int msbp)

Specifies the frequency of calls to the linear solver setup routine,  $msbp$  from §2.15.2.3.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **msbp** – the linear solver setup frequency.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.



**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Positive values of **msbp** specify the linear solver setup frequency. For example, an input of 1 means the setup function will be called every time step while an input of 2 means it will be called every other time step. If **msbp** is 0, the default value of 20 will be used. A negative value forces a linear solver step at each implicit stage.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.lsetup\_frequency”.

Added in version 6.1.0.

int **ARKodeSetJacEvalFrequency**(void \*arkode\_mem, long int msbj)

Specifies the number of steps after which the Jacobian information is considered out-of-date, *msbj* from §2.15.2.3.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **msbj** – the Jacobian re-computation or preconditioner update frequency.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK stepper unsupported** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

If *nstlj* is the step number at which the Jacobian information was last updated and *nst* is the current step number,  $nst - nstlj \geq msbj$  indicates that the Jacobian information will be updated during the next linear solver setup call.

As the Jacobian update frequency is only checked *within* calls to the linear solver setup routine, Jacobian information may be more than *msbj* steps old when updated depending on when a linear solver setup call occurs. See §2.15.2.3 for more information on when linear solver setups are performed.

Passing a value *msbj*  $\leq 0$  indicates to use the default value of 51.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [ARKodeSetLinearSolver\(\)](#).

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.jac\_eval\_frequency”.

Added in version 6.1.0.

## Optional inputs for matrix-based SUNLinearSolver modules

Optional input	Function name	Default
Jacobian function	<a href="#">ARKodeSetJacFn()</a>	DQ
Linear system function	<a href="#">ARKodeSetLinSysFn()</a>	internal
Mass matrix function	<a href="#">ARKodeSetMassFn()</a>	none
Enable or disable linear solution scaling	<a href="#">ARKodeSetLinearSolutionScaling()</a>	on

When using matrix-based linear solver modules, the ARKLS solver interface needs a function to compute an approximation to the Jacobian matrix  $J(t, y)$  or the linear system  $\mathcal{A}(t, y) = M(t) - \gamma J(t, y)$ .

For  $J(t, y)$ , the ARKLS interface is packaged with a routine that can approximate  $J$  if the user has selected either the [SUNMATRIX\\_DENSE](#) or [SUNMATRIX\\_BAND](#) objects. Alternatively, the user can supply a custom Jacobian function of type [ARKLSJacFn\(\)](#) – this is *required* when the user selects other matrix formats. To specify a user-supplied Jacobian function, ARKODE provides the function [ARKodeSetJacFn\(\)](#).

Alternatively, a function of type [ARKLSLinSysFn\(\)](#) can be provided to evaluate the matrix  $\mathcal{A}(t, y)$ . By default, ARKLS uses an internal linear system function leveraging the SUNMATRIX API to form the matrix  $\mathcal{A}(t, y)$  by combining the matrices  $M(t)$  and  $J(t, y)$ . To specify a user-supplied linear system function instead, ARKODE provides the function [ARKodeSetLinSysFn\(\)](#).

If the ODE system involves a non-identity mass matrix,  $M \neq I$ , matrix-based linear solver modules require a function to compute an approximation to the mass matrix  $M(t)$ . There is no default difference quotient approximation (for any matrix type), so this routine must be supplied by the user. This function must be of type [ARKLSMassFn\(\)](#), and should be set using the function [ARKodeSetMassFn\(\)](#).

In either case ( $J(t, y)$  versus  $\mathcal{A}(t, y)$  is supplied) the matrix information will be updated infrequently to reduce matrix construction and, with direct solvers, factorization costs. As a result the value of  $\gamma$  may not be current and a scaling factor is applied to the solution of the linear system to account for the lagged value of  $\gamma$ . See §10.2.1 for more details. The function [ARKodeSetLinearSolutionScaling\(\)](#) can be used to disable this scaling when necessary, e.g., when providing a custom linear solver that updates the matrix using the current  $\gamma$  as part of the solve.

The ARKLS interface passes the user data pointer to the Jacobian, linear system, and mass matrix functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian, linear system or mass matrix functions, without using global data in the program. The user data pointer may be specified through [ARKodeSetUserData\(\)](#).

int [ARKodeSetJacFn](#)(void \*arkode\_mem, [ARKLSJacFn](#) jac)

Specifies the Jacobian approximation routine to be used for the matrix-based solver with the ARKLS interface.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **jac** – name of user-supplied Jacobian approximation function.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This routine must be called after the ARKLS linear solver interface has been initialized through a call to [ARKodeSetLinearSolver\(\)](#).

By default, ARKLS uses an internal difference quotient function for the [SUNMATRIX\\_DENSE](#) and [SUNMATRIX\\_BAND](#) modules. If NULL is passed in for *jac*, this default is used. An error will occur if no *jac* is supplied when using other matrix types.

The function type [ARKLsJacFn\(\)](#) is described in §5.4.

Added in version 6.1.0.

int **ARKodeSetLinSysFn**(void \*arkode\_mem, [ARKLsLinSysFn](#) linsys)

Specifies the linear system approximation routine to be used for the matrix-based solver with the ARKLS interface.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **linsys** – name of user-supplied linear system approximation function.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This routine must be called after the ARKLS linear solver interface has been initialized through a call to [ARKodeSetLinearSolver\(\)](#).

By default, ARKLS uses an internal linear system function that leverages the SUNMATRIX API to form the system  $M - \gamma J$ . If NULL is passed in for *linsys*, this default is used.

The function type [ARKLsLinSysFn\(\)](#) is described in §5.4.

Added in version 6.1.0.

int **ARKodeSetMassFn**(void \*arkode\_mem, [ARKLsMassFn](#) mass)

Specifies the mass matrix approximation routine to be used for the matrix-based solver with the ARKLS interface.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **mass** – name of user-supplied mass matrix approximation function.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.

- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_MASSMEM\_NULL** – the mass matrix solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an argument had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

This routine must be called after the ARKLS mass matrix solver interface has been initialized through a call to [`ARKodeSetMassLinearSolver\(\)`](#).

Since there is no default difference quotient function for mass matrices, *mass* must be non-NULL.

The function type [`ARKLsMassFn\(\)`](#) is described in §5.4.

Added in version 6.1.0.

int **ARKodeSetLinearSolutionScaling**(void \*arkode\_mem, *sunbooleantype* onoff)

Enables or disables scaling the linear system solution to account for a change in  $\gamma$  in the linear system. For more details see §10.2.1.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **onoff** – flag to enable (SUNTRUE) or disable (SUNFALSE) scaling.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_ILL\_INPUT** – the attached linear solver is not matrix-based.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Linear solution scaling is enabled by default when a matrix-based linear solver is attached.

This routine will be called by [`ARKodeSetOptions\(\)`](#) when using the key “arkid.linear\_solution\_scaling”.

Added in version 6.1.0.

## Optional inputs for matrix-free SUNLinearSolver modules

Optional input	Function name	Default
<i>Jv</i> functions ( <i>jt看imes</i> and <i>jtsetup</i> )	<a href="#">ARKodeSetJacTimes()</a>	DQ, none
<i>Jv</i> DQ rhs function ( <i>jtimesRhsFn</i> )	<a href="#">ARKodeSetJacTimesRhsFn()</a>	fi
<i>Mv</i> functions ( <i>mtimes</i> and <i>mtsetup</i> )	<a href="#">ARKodeSetMassTimes()</a>	none, none

As described in §2.15.2, when solving the Newton linear systems with matrix-free methods, the ARKLS interface requires a *jt看imes* function to compute an approximation to the product between the Jacobian matrix  $J(t, y)$  and a vector  $v$ . The user can supply a custom Jacobian-times-vector approximation function, or use the default internal difference quotient function that comes with the ARKLS interface.

A user-defined Jacobian-vector function must be of type [ARKLSJacTimesVecFn](#) and can be specified through a call to [ARKodeSetJacTimes\(\)](#) (see §5.4 for specification details). As with the user-supplied preconditioner functions, the evaluation and processing of any Jacobian-related data needed by the user's Jacobian-times-vector function is done in the optional user-supplied function of type [ARKLSJacTimesSetupFn](#) (see §5.4 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.

int **ARKodeSetJacTimes**(void \*arkode\_mem, [ARKLSJacTimesSetupFn](#) jtsetup, [ARKLSJacTimesVecFn](#) jt看imes)

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

## Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **jtsetup** – user-defined Jacobian-vector setup function. Pass NULL if no setup is necessary.
- **jt看imes** – user-defined Jacobian-vector product function.

## Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARKLS\_SUNLS\_FAIL** – an error occurred when setting up the Jacobian-vector product in the SUNLinearSolver object used by the ARKLS interface.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

## Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The default is to use an internal finite difference quotient for *jt看imes* and to leave out *jtsetup*. If NULL is passed to *jt看imes*, these defaults are used. A user may specify non-NULL *jt看imes* and NULL *jtsetup* inputs.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [ARKodeSetLinearSolver\(\)](#).

The function types [ARKLSJacTimesSetupFn](#) and [ARKLSJacTimesVecFn](#) are described in §5.4.

Added in version 6.1.0.

When using the internal difference quotient the user may optionally supply an alternative implicit right-hand side function for use in the Jacobian-vector product approximation by calling `ARKodeSetJacTimesRhsFn()`. The alternative implicit right-hand side function should compute a suitable (and differentiable) approximation to the  $f^I$  function provided to `*StepCreate`. For example, as done in [38], the alternative function may use lagged values when evaluating a nonlinearity in  $f^I$  to avoid differencing a potentially non-differentiable factor. We note that in many instances this same  $f^I$  routine would also have been desirable for the nonlinear solver, in which case the user should specify this through calls to both `ARKodeSetJacTimesRhsFn()` and `ARKodeSetNlsRhsFn()`.

```
int ARKodeSetJacTimesRhsFn(void *arkode_mem, ARKRhsFn jtimesRhsFn)
```

Specifies an alternative implicit right-hand side function for use in the internal Jacobian-vector product difference quotient approximation.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **jtimesRhsFn** – the name of the C function (of type `ARKRhsFn()`) defining the alternative right-hand side function.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARK stepper unsupported** – implicit solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The default is to use the implicit right-hand side function provided to `*StepCreate` in the internal difference quotient. If the input implicit right-hand side function is NULL, the default is used.

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

Added in version 6.1.0.

Similarly, if a problem involves a non-identity mass matrix,  $M \neq I$ , then matrix-free solvers require a *mtimes* function to compute an approximation to the product between the mass matrix  $M(t)$  and a vector  $v$ . This function must be user-supplied since there is no default value, it must be of type `ARKLsMassTimesVecFn()`, and can be specified through a call to the `ARKodeSetMassTimes()` routine. Similarly to the user-supplied preconditioner functions, any evaluation and processing of any mass matrix-related data needed by the user's mass-matrix-times-vector function may be done in an optional user-supplied function of type `ARKLsMassTimesSetupFn` (see §5.4 for specification details).

```
int ARKodeSetMassTimes(void *arkode_mem, ARKLsMassTimesSetupFn mtsetup, ARKLsMassTimesVecFn
    mtimes, void *mtimes_data)
```

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

#### Parameters

- **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 values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_MASSMEM\_NULL** – the mass matrix solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARKLS\_SUNLS\_FAIL** – an error occurred when setting up the mass-matrix-vector product in the `SUNLinearSolver` object used by the ARKLS interface.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

There is no default finite difference quotient for *mtimes*, so if using the ARKLS mass matrix solver interface with NULL-valued `SUNMATRIX` input *M*, and this routine is called with NULL-valued *mtimes*, an error will occur. A user may specify NULL for *mtsetup*.

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

The function types `ARKLSMassTimesSetupFn` and `ARKLSMassTimesVecFn` are described in §5.4.

Added in version 6.1.0.

### Optional inputs for iterative `SUNLinearSolver` modules

Optional input	Function name	Default
Newton preconditioning functions	<code>ARKodeSetPreconditioner()</code>	NULL, NULL
Mass matrix preconditioning functions	<code>ARKodeSetMassPreconditioner()</code>	NULL, NULL
Newton linear and nonlinear tolerance ratio	<code>ARKodeSetEpsLin()</code>	0.05
Mass matrix linear and nonlinear tolerance ratio	<code>ARKodeSetMassEpsLin()</code>	0.05
Newton linear solve tolerance conversion factor	<code>ARKodeSetLSNormFactor()</code>	vector length
Mass matrix linear solve tolerance conversion factor	<code>ARKodeSetMassLSNormFactor()</code>	vector length

As described in §2.15.2, when using an iterative linear solver the user may supply a preconditioning operator to aid in solution of the system. This operator consists of two user-supplied functions, *psetup* and *psolve*, that are supplied to ARKODE using either the function `ARKodeSetPreconditioner()` (for preconditioning the Newton system), or the function `ARKodeSetMassPreconditioner()` (for preconditioning the mass matrix system). The *psetup* function supplied to these routines should handle evaluation and preprocessing of any Jacobian or mass-matrix data needed by the user's preconditioner solve function, *psolve*. The user data pointer received through `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.

Also, as described in §2.15.3.2, the ARKLS interface requires that iterative linear solvers stop when the norm of the preconditioned residual satisfies

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}$$

where the default  $\epsilon_L = 0.05$  may be modified by the user through the [ARKodeSetEpsLin\(\)](#) function.

int **ARKodeSetPreconditioner**(void \*arkode\_mem, [ARKLsPrecSetupFn](#) psetup, [ARKLsPrecSolveFn](#) psolve)

Specifies the user-supplied preconditioner setup and solve functions.

#### Parameters

- **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 values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARKLS\_SUNLS\_FAIL** – an error occurred when setting up preconditioning in the SUNLinearSolver object used by the ARKLS interface.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

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

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [ARKodeSetLinearSolver\(\)](#).

Both of the function types [ARKLsPrecSetupFn\(\)](#) and [ARKLsPrecSolveFn\(\)](#) are described in §5.4.

Added in version 6.1.0.

int **ARKodeSetMassPreconditioner**(void \*arkode\_mem, [ARKLsMassPrecSetupFn](#) psetup, [ARKLsMassPrecSolveFn](#) psolve)

Specifies the mass matrix preconditioner setup and solve functions.

#### Parameters

- **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 values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARKLS\_SUNLS\_FAIL** – an error occurred when setting up preconditioning in the `SUNLinearSolver` object used by the ARKLS interface.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

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

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

Both of the function types `ARKLsMassPrecSetupFn()` and `ARKLsMassPrecSolveFn()` are described in §5.4.

Added in version 6.1.0.

int **ARKodeSetEpsLin**(void \*arkode\_mem, *sunrealtype* eplifac)

Specifies the factor  $\epsilon_L$  by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the linear iteration.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **eplifac** – linear convergence safety factor.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Passing a value  $eplifac \leq 0$  indicates to use the default value of 0.05.

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

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.eps\_lin”.

Added in version 6.1.0.

int **ARKodeSetMassEpsLin**(void \*arkode\_mem, *sunrealtype* eplifac)

Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the mass matrix linear iteration.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **eplifac** – linear convergence safety factor.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_MASSMEM\_NULL** – the mass matrix solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to [ARKodeSetMassLinearSolver\(\)](#).

Passing a value  $eplifac \leq 0$  indicates to use the default value of 0.05.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.mass\_eps\_lin”.

Added in version 6.1.0.

Since iterative linear solver libraries typically consider linear residual tolerances using the  $L_2$  norm, whereas ARKODE focuses on errors measured in the WRMS norm (2.24), the ARKLS interface internally converts between these quantities when interfacing with linear solvers,

$$\text{tol}_{L_2} = \text{nrmfac} \text{ tol}_{WRMS}. \quad (5.1)$$

Prior to the introduction of [N\\_VGetLength\(\)](#) in SUNDIALS v5.0.0 the value of *nrmfac* was computed using the vector dot product. Now, the functions [ARKodeSetLSNormFactor\(\)](#) and [ARKodeSetMassLSNormFactor\(\)](#) allow for additional user control over these conversion factors.

int **ARKodeSetLSNormFactor**(void \*arkode\_mem, *sunrealtype* nrmfac)

Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for Newton linear system solves.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nrmfac** – the norm conversion factor. If *nrmfac* is:  
 $> 0$  then the provided value is used.  
 $= 0$  then the conversion factor is computed using the vector length i.e.,  $\text{nrmfac} = \text{sqrt}(\text{N\_VGetLength}(y))$  (default).

$< 0$  then the conversion factor is computed using the vector dot product i.e.,  $\text{nrmfac} = \sqrt{\text{N\_VDotProd}(\mathbf{v}, \mathbf{v})}$  where all the entries of  $\mathbf{v}$  are one.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [ARKodeSetLinearSolver\(\)](#).

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.ls\_norm\_factor”.

Added in version 6.1.0.

int **ARKodeSetMassLSNormFactor**(void \*arkode\_mem, *sunrealtype* nrmfac)

Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for mass matrix linear system solves.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nrmfac** – the norm conversion factor. If *nrmfac* is:
  - $> 0$  then the provided value is used.
  - $= 0$  then the conversion factor is computed using the vector length i.e.,  $\text{nrmfac} = \sqrt{\text{N\_VGetLength}(\mathbf{y})}$  (*default*).
  - $< 0$  then the conversion factor is computed using the vector dot product i.e.,  $\text{nrmfac} = \sqrt{\text{N\_VDotProd}(\mathbf{v}, \mathbf{v})}$  where all the entries of  $\mathbf{v}$  are one.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to [ARKodeSetMassLinearSolver\(\)](#).

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.mass\_ls\_norm\_factor”.

Added in version 6.1.0.

### 5.3.8.5 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 §2.16.

Optional input	Function name	Default
Direction of zero-crossings to monitor	<a href="#"><code>ARKodeSetRootDirection()</code></a>	both
Disable inactive root warnings	<a href="#"><code>ARKodeSetNoInactiveRootWarn()</code></a>	enabled

int **ARKodeSetRootDirection**(void \*arkode\_mem, int \*rootdir)

Specifies the direction of zero-crossings to be located and returned.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **rootdir** – state array of length *nrtfn*, the number of root functions  $g_i$  (the value of *nrtfn* was supplied in the call to [`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 values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

#### Note

The default behavior is to monitor for both zero-crossing directions.

Added in version 6.1.0.

int **ARKodeSetNoInactiveRootWarn**(void \*arkode\_mem)

Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

#### Note

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.

This routine will be called by [`ARKodeSetOptions\(\)`](#) when using the key “`arkid.no_inactive_root_warn`”.

Added in version 6.1.0.

### 5.3.8.6 Pre-step, Post-step, Pre-RHS, and Post-processing optional inputs (ADVANCED)

ARKODE provides multiple options for user-supplied callback routines that can be called at various times within the time-stepping process. Each of these callback functions has a similar structure, wherein the callback function will be provided with the current time, current solution, and the *user\_data* structure that was provided to `ARKodeSetUserData()`; some functions are also provided the current time step counter, or even a counter indicating how many times this step has been attempted previously. More specifically, users may provide callback functions for the following events within a time step:

- just prior to starting a time step attempt (`ARKodeSetPreStepFn()`),
- at the end of a successful time step (`ARKodeSetPostStepFn()`),
- just prior to evaluating user-provided right-hand side (RHS) functions (`ARKodeSetPreRhsFn()`),
- immediately after each stage is completed within a time step (`ARKodeSetPostprocessStageFn()`), and
- immediately after computing the new step but before the step is accepted or rejected (`ARKodeSetPostprocessStepFn()`).

For users who wish to perform different actions at individual internal stages within an ARKODE method, they may obtain the current stage index by calling `ARKodeGetStageIndex()` in their stage-level callback routines provided to `ARKodeSetPreRhsFn()` and `ARKodeSetPostprocessStageFn()`.

The specific ordering of these functions within a given step depends on whether each stage is explicit (as in ERKStep) or implicit (as in ARKStep or MRISStep). Denoting the most-recent “saved” time step as  $(t_n, y_n)$ , the time-evolving temporary state within a step as  $(t_{cur}, y_{cur})$ , the functions provided to the five above functions as PreStep, PostStep, PreRHS, PostprocessStage, and PostprocessStep, and denoting the IVP right hand side function as RHS, then the flow of a 3-stage explicit method would proceed as:

0. Initialize attempt counter to 0
1. Call PreStep with  $(t_n, y_n)$
2. Stage 0
  - a. If this is not the first step and the method is FSAL (First Same As Last – the last stage of the prior step is the current solution and the first stage is explicit) or the RHS has already been computed at  $(t_n, y_n)$ , proceed to stage 1.
  - b. Call PreRHS with  $(t_n, y_n)$
  - c. Evaluate RHS at  $(t_n, y_n)$
3. Stage 1
  - a. Compute the stage solution  $(t_{cur}, y_{cur})$
  - b. Call PostprocessStage with  $(t_{cur}, y_{cur})$
  - c. Call PreRHS with  $(t_{cur}, y_{cur})$
  - d. Evaluate RHS at  $(t_{cur}, y_{cur})$
4. Stage 2
  - a. Compute the stage solution  $(t_{cur}, y_{cur})$
  - b. If the method is FSAL call PostprocessStep with  $(t_{cur}, y_{cur})$ , else call PostprocessStage with  $(t_{cur}, y_{cur})$
  - c. Call PreRHS with  $(t_{cur}, y_{cur})$

- d. Evaluate RHS at  $(t_{cur}, y_{cur})$
5. If the method is not FSAL, compute the new time step solution  $(t_{cur}, y_{cur})$  and call `PostprocessStep` with  $(t_{cur}, y_{cur})$
6. Check the local error
  - a. If the step is successful then call `PostStep` with  $(t_{cur}, y_{cur})$ , determine the next internal step size  $h_n$ , and update  $(t_n, y_n) \leftarrow (t_{cur}, y_{cur})$
  - b. Else rewind  $(t_{cur}, y_{cur}) \leftarrow (t_n, y_n)$ , increment the `attempt` counter, determine the next internal step size  $h_n$ , and return to step 1

Alternately, the flow of a 3-stage method that must perform a solve of some sort for each stage (i.e., a DIRK or ARK method in `ARKStep`, or a multirate method with `MRIstep`) would proceed as follows. Here, we show the implicit-explicit approach since that also shows the relationship between both the implicit right-hand side function `RHS_i` and the explicit right-hand side function `RHS_e`:

0. Initialize `attempt` counter to 0
1. Call `PreStep` with  $(t_n, y_n)$
2. Stage 0
  - a. If the first stage is explicit:
    - i. If this is not the first step and the method is FSAL or `RHS_i` and `RHS_e` have already been computed at  $(t_n, y_n)$ , proceed to stage 1.
    - ii. Call `PreRHS` with  $(t_n, y_n)$
    - iii. Evaluate `RHS_i` and `RHS_e` at  $(t_n, y_n)$
  - b. Else the first stage is implicit:
    - i. Solve the implicit system, calling `PreRHS` and then `RHS_i` with  $(t_{cur}, y_{cur})$  at each solver iteration; at the end of this iteration  $(t_{cur}, y_{cur})$  holds the updated stage solution
    - ii. Call `PostprocessStage` with  $(t_{cur}, y_{cur})$
    - iii. Call `PreRHS` with  $(t_{cur}, y_{cur})$
    - iv. Evaluate `RHS_i` and then `RHS_e` at  $(t_{cur}, y_{cur})$
3. Stage 1
  - a. Solve the implicit system, calling `PreRHS` and then `RHS_i` with  $(t_{cur}, y_{cur})$  at each solver iteration; at the end of this iteration  $(t_{cur}, y_{cur})$  holds the updated stage solution
  - b. Call `PostprocessStage` with  $(t_{cur}, y_{cur})$
  - c. Call `PreRHS` with  $(t_{cur}, y_{cur})$
  - d. Evaluate `RHS_i` and then `RHS_e` at  $(t_{cur}, y_{cur})$
4. Stage 2
  - a. Solve implicit system, calling `PreRHS` and then `RHS_i` with  $(t_{cur}, y_{cur})$  at each solver iteration; at the end of this iteration  $(t_{cur}, y_{cur})$  holds the updated stage solution
  - b. If the method is stiffly accurate, call `PostprocessStep` with  $(t_{cur}, y_{cur})$ , else call `PostprocessStage` with  $(t_{cur}, y_{cur})$ ,
  - c. Call `PreRHS` with  $(t_{cur}, y_{cur})$
  - d. Evaluate `RHS_i` and then `RHS_e` at  $(t_{cur}, y_{cur})$

5. If the method is not stiffly accurate, compute the new time step solution  $(t_{cur}, y_{cur})$  and call `PostprocessStep` with  $(t_{cur}, y_{cur})$
6. Check the local error.
  - a. If the step is successful then call `PostStep` with  $(t_{cur}, y_{cur})$ , determine the next internal step size  $h_n$ , and update  $(t_n, y_n) \leftarrow (t_{cur}, y_{cur})$
  - b. Else rewind  $(t_{cur}, y_{cur}) \leftarrow (t_n, y_n)$ , increment `attempt` counter, determine the next internal step size  $h_n$ , and return to step 1

We consider these functions as “advanced” because of their danger, although the callback functions are provided with the internally-evolving state, users should **not** adjust entries of this state vector, since doing so will destroy all theoretical guarantees of solution accuracy and numerical stability. The only “supported” approach for user modifications to the state vector is if this occurs between calls to `ARKodeEvolve()`, and if the user calls `ARKodeReset()` after every modification to the state vector so that ARKODE can reset its saved solution.

Optional input	Function name	Default
Set pre time step function	<code>ARKodeSetPreStepFn()</code>	NULL
Set post time step function	<code>ARKodeSetPostStepFn()</code>	NULL
Set pre right-hand side function	<code>ARKodeSetPreRhsFn()</code>	NULL
Set stage postprocessing function	<code>ARKodeSetPostprocessStageFn()</code>	NULL
Set time step postprocessing function	<code>ARKodeSetPostprocessStepFn()</code>	NULL

int **ARKodeSetPreStepFn**(void \*arkode\_mem, *ARKPreStepFn* prestep\_fn)

[ADVANCED] Provide a function to be called before each step attempt.

The attached function allows users to set up auxiliary data structures that only need to be updated at the start of a step and can be reused within the time step (e.g., in their right-hand side function(s)).

#### Danger

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **prestep\_fn** – the user-supplied function to call. A NULL input function disables calling a prestep function.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.7.0.

int **ARKodeSetPostStepFn**(void \*arkode\_mem, *ARKPostStepFn* poststep\_fn)

[ADVANCED] Provide a function to be called following each successful time step.

The attached function allows users to compute relevant diagnostic information after each step.

**Danger**

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **poststep\_fn** – the user-supplied function to call. A NULL input function disables calling a poststep function.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL.

Added in version 6.7.0: This function replaces the undocumented [\*ARKodeSetPostprocessStepFn\(\)\*](#) used in earlier versions for attaching a function to be called after each successful step.

int **ARKodeSetPreRhsFn**(void \*arkode\_mem, [\*ARKPreRhsFn\*](#) prerhs\_fn)

[ADVANCED] Provides a function to be called prior to evaluating user-provided right-hand side (RHS) functions. For partitioned methods with multiple RHS functions (e.g., ARKStep or MRISStep), when multiple RHS functions will be called in succession with identical inputs, this function is called only once prior to the RHS function evaluations.

The attached function allows users to set up auxiliary data structures that will be used within the RHS evaluations (e.g., MPI communication to fill and send exchange buffers).

**Danger**

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **prerhs\_fn** – the user-supplied function to call. A NULL input function disables calling a pre-RHS function.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL.

Added in version 6.7.0.

int **ARKodeSetPostprocessStepFn**(void \*arkode\_mem, [\*ARKPostProcessFn\*](#) ProcessStep)

[ADVANCED] Provides a function to be called immediately after computing a new step but before the step is accepted/rejected.



**Danger**

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ProcessStep** – the user-supplied function to call. A NULL input function disables step postprocessing.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.7.0: This function existed in earlier versions as an undocumented feature and the attached function was called after each successful step. Starting with version 6.7.0, use [ARKodeSetPostStepFn\(\)](#) to attach a function to be called after each successful step.

**Warning**

This function is currently incompatible with discrete adjoint capabilities in ARKODE ([ARKodeSetAdjointCheckpointScheme\(\)](#) and [ARKodeSetAdjointCheckpointIndex\(\)](#)).

int **ARKodeSetPostprocessStageFn**(void \*arkode\_mem, [ARKPostProcessFn](#) ProcessStage)

[ADVANCED] Provides a function to be called immediately after each stage is completed within ARKODE's multi-stage methods.

**Danger**

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ProcessStage** – the user-supplied function to call. A NULL input function disables stage postprocessing.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.7.0: This function existed in earlier versions as an undocumented feature.

**Warning**

This function is currently incompatible with discrete adjoint capabilities in ARKODE ([ARKodeSetAdjointCheckpointScheme\(\)](#) and [ARKodeSetAdjointCheckpointIndex\(\)](#)).

### 5.3.9 Interpolated output function

An optional function [ARKodeGetDky\(\)](#) is available to obtain additional values of solution-related quantities. This function should only be called after a successful return from [ARKodeEvolve\(\)](#), as it provides interpolated values either of  $y$  or of its derivatives (up to the 5th derivative) interpolated to any value of  $t$  in the last internal step taken by [ARKodeEvolve\(\)](#). Internally, this “dense output” or “continuous extension” algorithm is identical to the algorithm used for the maximum order implicit predictors, described in §2.15.5.2, except that derivatives of the polynomial model may be evaluated upon request.

int **ARKodeGetDky**(void \*arkode\_mem, *sunrealtype* t, int k, *N\_Vector* dky)

Computes the  $k$ -th derivative of the function  $y$  at the time  $t$ , i.e.  $y^{(k)}(t)$ , 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. This routine uses an interpolating polynomial of degree  $\min(\text{degree}, 5)$ , where *degree* is the argument provided to [ARKodeSetInterpolantDegree\(\)](#). The user may request  $k$  in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$  where  $kmax$  depends on the choice of interpolation module. For Hermite interpolants  $kmax = 5$  and for Lagrange interpolants  $kmax = 3$ .

**Parameters**

- **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 values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_BAD\_K** –  $k$  is not in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$ .
- **ARK\_BAD\_T** –  $t$  is not in the interval  $[t_n - h_n, t_n]$ .
- **ARK\_BAD\_DKY** – the *dky* vector was NULL.
- **ARK\_MEM\_NULL** – *arkode\_mem* was NULL.

**Note**

It is only legal to call this function after a successful return from [ARKodeEvolve\(\)](#).

A user may access the values  $t_n$  and  $h_n$  via the functions [ARKodeGetCurrentTime\(\)](#) and [ARKodeGetLastStep\(\)](#), respectively.

Added in version 6.1.0.

### 5.3.10 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 §5.3.10.1,
2. ARKODE implicit solver output routines are in §5.3.10.2,
3. Output routines regarding root-finding results are in §5.3.10.3,
4. Linear solver output routines are in §5.3.10.4 and
5. General usability routines (e.g. to print the current ARKODE parameters, or output the current Butcher table(s)) are in §5.3.10.5.

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 ARKODE. 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* (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, and hence a lower ratio.
- The ratio *nsteps/step\_attempts* can measure the quality of the time step adaptivity algorithm, since a poor algorithm will result in more failed steps, and hence a lower ratio.

It is therefore recommended that users retrieve and output these statistics following each run, and take some time to investigate alternate solver options that will be more optimal for their particular problem of interest.

## 5.3.10.1 Main solver optional output functions

Optional output	Function name
Size of ARKODE real and integer workspaces	<i>ARKodeGetWorkSpace()</i>
Cumulative number of internal steps	<i>ARKodeGetNumSteps()</i>
Actual initial time step size used	<i>ARKodeGetActualInitStep()</i>
Step size used for the last successful step	<i>ARKodeGetLastStep()</i>
Step size to be attempted on the next step	<i>ARKodeGetCurrentStep()</i>
Integration direction, e.g., forward or backward	<i>ARKodeGetStepDirection()</i>
Last saved time reached by the solver	<i>ARKodeGetLastTime()</i>
Last saved solution reached by the solver	<i>ARKodeGetLastState()</i>
Current internal time reached by the solver	<i>ARKodeGetCurrentTime()</i>
Current internal solution reached by the solver	<i>ARKodeGetCurrentState()</i>
Current $\gamma$ value used by the solver	<i>ARKodeGetCurrentGamma()</i>
Suggested factor for tolerance scaling	<i>ARKodeGetTolScaleFactor()</i>
Error weight vector for state variables	<i>ARKodeGetErrWeights()</i>
Residual weight vector	<i>ARKodeGetResWeights()</i>
Single accessor to many statistics at once	<i>ARKodeGetStepStats()</i>
Print all statistics	<i>ARKodePrintAllStats()</i>
Name of constant associated with a return flag	<i>ARKodeGetReturnFlagName()</i>
No. of explicit stability-limited steps	<i>ARKodeGetNumExpSteps()</i>
No. of accuracy-limited steps	<i>ARKodeGetNumAccSteps()</i>
No. of attempted steps	<i>ARKodeGetNumStepAttempts()</i>
No. of RHS evaluations	<i>ARKodeGetNumRhsEvals()</i>
No. of local error test failures that have occurred	<i>ARKodeGetNumErrTestFails()</i>
No. of failed steps due to a nonlinear solver failure	<i>ARKodeGetNumStepSolveFails()</i>
Estimated local truncation error vector	<i>ARKodeGetEstLocalErrors()</i>
Number of constraint test failures	<i>ARKodeGetNumConstrFails()</i>
Retrieve a pointer for user data	<i>ARKodeGetUserData()</i>
Retrieve the accumulated temporal error estimate	<i>ARKodeGetAccumulatedError()</i>
Current stage index, and total number of stages	<i>ARKodeGetStageIndex()</i>

int **ARKodeGetWorkSpace**(void \*arkode\_mem, long int \*lenrw, long int \*leniw)

Returns the ARKODE real and integer workspace sizes.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **lenrw** – the number of `sunrealtype` values in the ARKODE workspace.
- **leniw** – the number of integer values in the ARKODE workspace.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.1.0.

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

int **ARKodeGetNumSteps**(void \*arkode\_mem, long int \*nsteps)

Returns the cumulative number of internal steps taken by the solver (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nsteps** – number of steps taken in the solver.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.1.0.

int **ARKodeGetActualInitStep**(void \*arkode\_mem, *sunrealtype* \*hinused)

Returns the value of the integration step size used on the first step.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **hinused** – actual value of initial step size.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

**Note**

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.

Added in version 6.1.0.

int **ARKodeGetLastStep**(void \*arkode\_mem, *sunrealtype* \*hlast)

Returns the integration step size taken on the last successful internal step.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **hlast** – step size taken on the last internal step.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.1.0.

int **ARKodeGetCurrentStep**(void \*arkode\_mem, *sunrealtype* \*hcur)

Returns the integration step size to be attempted on the next internal step.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **hcur** – step size to be attempted on the next internal step.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.

- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.1.0.

int **ARKodeGetStepDirection**(void \*arkode\_mem, *sunrealtype* \*stepdir)

Returns the direction of integration that will be used on the next internal step.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **stepdir** – a positive number if integrating forward, a negative number if integrating backward, or zero if the direction has not been set.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.2.0.

int **ARKodeGetLastTime**(void \*arkode\_mem, *sunrealtype* \*tn)

Returns the last saved time reached by the solver.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **tn** – last saved time reached.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.7.0.

int **ARKodeGetLastState**(void \*arkode\_mem, *N\_Vector* \*yn)

Returns the last saved solution reached by the solver.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **yn** – last saved solution.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

**Danger**

Users should exercise extreme caution when using this function, as altering values of *yn* may lead to undesirable behavior, depending on the particular use case and on when this routine is called.

Added in version 6.7.0.

int **ARKodeGetCurrentTime**(void \*arkode\_mem, *sunrealtype* \*tcur)

Returns the current internal time reached by the solver.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **tcur** – current internal time reached.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.1.0.

int **ARKodeGetCurrentState**(void \*arkode\_mem, *N\_Vector* \*ycur)

Returns the current internal solution reached by the solver.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ycur** – current internal solution.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

**Note**

Users should exercise extreme caution when using this function, as altering values of *ycur* may lead to undesirable behavior, depending on the particular use case and on when this routine is called.

Added in version 6.1.0.

int **ARKodeGetCurrentGamma**(void \*arkode\_mem, *sunrealtype* \*gamma)

Returns the current internal value of  $\gamma$  used in the implicit solver Newton matrix (see equation (2.47)).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **gamma** – current step size scaling factor in the Newton system.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Added in version 6.1.0.

int **ARKodeGetTolScaleFactor**(void \*arkode\_mem, *sunrealtype* \*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.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **tolsfac** – suggested scaling factor for user-supplied tolerances.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.1.0.

int **ARKodeGetErrWeights**(void \*arkode\_mem, *N\_Vector* eweight)

Returns the current error weight vector.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **eweight** – solution error weights at the current time.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

**Note**

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

Added in version 6.1.0.

int **ARKodeGetResWeights**(void \*arkode\_mem, *N\_Vector* rweight)

Returns the current residual weight vector.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **rweight** – residual error weights at the current time.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

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

Added in version 6.1.0.



```
int ARKodeGetStepStats(void *arkode_mem, long int *nsteps, sunrealtype *hinused, sunrealtype *hlast,
                       sunrealtype *hcur, sunrealtype *tcur)
```

Returns many of the most useful optional outputs in a single call.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nsteps** – number of steps taken in the solver.
- **hinused** – actual value of initial step size.
- **hlast** – step size taken on the last internal step.
- **hcur** – step size to be attempted on the next internal step.
- **tcur** – current internal time reached.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

Added in version 6.1.0.

```
int ARKodePrintAllStats(void *arkode_mem, FILE *outfile, SUNOutputFormat fmt)
```

Outputs all of the integrator, nonlinear solver, linear solver, and other statistics.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **outfile** – pointer to output file.
- **fmt** – the output format:
  - *SUN\_OUTPUTFORMAT\_TABLE* – prints a table of values
  - *SUN\_OUTPUTFORMAT\_CSV* – prints a comma-separated list of key and value pairs e.g., key1,value1,key2,value2,...

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – an invalid formatting option was provided.

#### Note

The Python module `tools/suntools` provides utilities to read and output the data from a SUNDIALS CSV output file using the key and value pair format.

Added in version 6.1.0.

```
char *ARKodeGetReturnFlagName(long int flag)
```

Returns the name of the ARKODE constant corresponding to *flag*. See *ARKODE Constants*.

#### Parameters

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

**Returns**

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

**Warning**

The user is responsible for freeing the returned string.

Added in version 6.1.0.

int **ARKodeGetNumExpSteps**(void \*arkode\_mem, long int \*expsteps)

Returns the cumulative number of stability-limited steps taken by the solver (so far). If the combination of the maximum number of stages and the current time step size in the LSRKStep module will not allow for a stable step, the counter also accounts for such returns.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **expsteps** – number of stability-limited steps taken in the solver.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

Added in version 6.1.0.

int **ARKodeGetNumAccSteps**(void \*arkode\_mem, long int \*accsteps)

Returns the cumulative number of accuracy-limited steps taken by the solver (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **accsteps** – number of accuracy-limited steps taken in the solver.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

Added in version 6.1.0.

int **ARKodeGetNumStepAttempts**(void \*arkode\_mem, long int \*step\_attempts)

Returns the cumulative number of steps attempted by the solver (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **step\_attempts** – number of steps attempted by solver.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

Added in version 6.1.0.

int **ARKodeGetNumRhsEvals**(void \*arkode\_mem, int partition\_index, long int \*num\_rhs\_evals)

Returns the number of calls to the user's right-hand side function (so far). For implicit methods or methods with an implicit partition, the count does not include calls made by a linear solver or preconditioner.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **num\_partition** – the right-hand side partition index:
  - For ERKStep, 0 corresponds to  $f(t, y)$
  - For ARKStep, 0 corresponds to  $f^E(t, y)$  and 1 to  $f^I(t, y)$
  - For MRISStep, 0 corresponds to  $f^E(t, y)$  and 1 to  $f^I(t, y)$
  - For SPRKStep, 0 corresponds to  $f_1(t, p)$  and 1 to  $f_2(t, q)$

A negative index will return the sum of the evaluations for each partition.

- **num\_rhs\_evals** – the number of right-hand side evaluations.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – if arkode\_mem was NULL.
- **ARK\_ILL\_INPUT** – if num\_partition was invalid for the stepper or num\_rhs\_evals was NULL

Added in version 6.2.0.

int **ARKodeGetNumErrTestFails**(void \*arkode\_mem, long int \*netfails)

Returns the number of local error test failures that have occurred (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **netfails** – number of error test failures.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

Added in version 6.1.0.

int **ARKodeGetNumStepSolveFails**(void \*arkode\_mem, long int \*ncnf)

Returns the number of failed steps due to a nonlinear solver failure (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ncnf** – number of step failures.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – implicit solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Added in version 6.1.0.

int **ARKodeGetEstLocalErrors**(void \*arkode\_mem, *N\_Vector* ele)

Returns the vector of estimated local truncation errors for the current step.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ele** – vector of estimated local truncation errors.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

**Note**

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

The values returned in *ele* are valid only after a successful call to [ARKodeEvolve\(\)](#) (i.e., it 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 WRMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

Added in version 6.1.0.

int **ARKodeGetNumConstrFails**(void \*arkode\_mem, long int \*nconstrfails)

Returns the cumulative number of constraint test failures (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nconstrfails** – number of constraint test failures.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – adaptive step sizes are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support temporal adaptivity.

Added in version 6.1.0.

int **ARKodeGetUserData**(void \*arkode\_mem, void \*\*user\_data)

Returns the user data pointer previously set with [ARKodeSetUserData\(\)](#).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **user\_data** – memory reference to a user data pointer.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

Added in version 6.1.0.

int **ARKodeGetAccumulatedError**(void \*arkode\_mem, *sunrealtype* \*accum\_error)

Returns the accumulated temporal error estimate.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **accum\_error** – pointer to accumulated error estimate.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_WARNING** – accumulated error estimation is currently disabled.
- **ARK\_STEPPER\_UNSUPPORTED** – temporal error estimation is not supported by the current time-stepping module.

Added in version 6.2.0.

int **ARKodeGetStageIndex**(void \*arkode\_mem, int \*stage, int \*max\_stages)

Returns the index of the current stage (0-based) and the total number of stages in the method i.e., for an  $s$ -stage method  $stage \in 0, \dots, s - 1$  and  $max\_stages = s$ .

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **stage** – pointer to storage for the current stage index.
- **max\_stages** – pointer to storage for the number of stages in the method.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – stage indexing is not supported by the current time-stepping module.

#### Note

For temporally adaptive computations in MRISStep, the “embedding” stage is indicated using *stage* **equal to** *max\_stages*.

For the methods in LSRKStep the number of “stages” in each method,  $s$ , corresponds with the number of solution updates, and thus this is one larger than what  $s$  denotes for explicit Runge–Kutta methods. Thus when calling *ARKodeGetStageIndex* while using LSRKStep, *stage* will range from 0 to  $s$  (inclusive), and *max\_stages* will be  $s+1$ .

Added in version 6.7.0.

### 5.3.10.2 Implicit solver optional output functions

Optional output	Function name
Computes state given a correction	<a href="#"><i>ARKodeComputeState()</i></a>
Access data to compute the nonlin. sys. function	<a href="#"><i>ARKodeGetNonlinearSystemData()</i></a>
No. of calls to linear solver setup function	<a href="#"><i>ARKodeGetNumLinSolvSetups()</i></a>
No. of nonlinear solver iterations	<a href="#"><i>ARKodeGetNumNonlinSolvIters()</i></a>
No. of nonlinear solver iterations	<a href="#"><i>ARKodeGetNumNonlinSolvIters()</i></a>
No. of nonlinear solver convergence failures	<a href="#"><i>ARKodeGetNumNonlinSolvConvFails()</i></a>
Single accessor to all nonlinear solver statistics	<a href="#"><i>ARKodeGetNonlinSolvStats()</i></a>

int **ARKodeGetNumLinSolvSetups**(void \*arkode\_mem, long int \*nlinsetups)

Returns the number of calls made to the linear solver’s setup routine (so far).

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nlinsetups** – number of linear solver setup calls made.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.

- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumNonlinSolvIters**(void \*arkode\_mem, long int \*nniters)

Returns the number of nonlinear solver iterations performed (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nniters** – number of nonlinear iterations performed.

**Return values**

- **ARK\_STEPPER\_UNSUPPORTED** – nonlinear solvers are not supported by the current time-stepping module.
- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NLS\_OP\_ERR** – the SUNNONLINSOL object returned a failure flag.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumNonlinSolvConvFails**(void \*arkode\_mem, long int \*nncfails)

Returns the number of nonlinear solver convergence failures that have occurred (so far).

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nncfails** – number of nonlinear convergence failures.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – nonlinear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNonlinSolvStats**(void \*arkode\_mem, long int \*nniters, long int \*nncfails)

Returns all of the nonlinear solver statistics in a single call.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nniters** – number of nonlinear iterations performed.
- **nncfails** – number of nonlinear convergence failures.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_NLS\_OP\_ERR** – the SUNNONLINSOL object returned a failure flag.
- **ARK\_STEPPER\_UNSUPPORTED** – nonlinear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the nonlinear solver object; the counters are reset whenever a new nonlinear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

**5.3.10.3 Rootfinding optional output functions**

Optional output	Function name
Array showing roots found	<a href="#"><i>ARKodeGetRootInfo()</i></a>
No. of calls to user root function	<a href="#"><i>ARKodeGetNumGEvals()</i></a>

int **ARKodeGetRootInfo**(void \*arkode\_mem, int \*rootsfound)

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

**Parameters**

- **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 (the value of *nrtfn* was supplied in the call to [\*ARKodeRootInit\(\)\*](#)). For  $i = 0 \dots nrtfn-1$ , *rootsfound*[*i*] is nonzero if  $g_i$  has a root, and 0 if not.



**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

**Note**

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  $g_i$ .

Added in version 6.1.0.

int **ARKodeGetNumGEvals**(void \*arkode\_mem, long int \*ngevals)

Returns the cumulative number of calls made to the user's root function  $g$ .

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ngevals** – number of calls made to  $g$  so far.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.

Added in version 6.1.0.

**5.3.10.4 Linear solver interface optional output functions**

A variety of optional outputs are available from the ARKLS interface, as listed in the following table and elaborated below. We 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
Stored Jacobian of the ODE RHS function	<a href="#">ARKodeGetJac()</a>
Time at which the Jacobian was evaluated	<a href="#">ARKodeGetJacTime()</a>
Step number at which the Jacobian was evaluated	<a href="#">ARKodeGetJacNumSteps()</a>
Size of real and integer workspaces	<a href="#">ARKodeGetLinWorkSpace()</a>
No. of Jacobian evaluations	<a href="#">ARKodeGetNumJacEvals()</a>
No. of preconditioner evaluations	<a href="#">ARKodeGetNumPrecEvals()</a>
No. of preconditioner solves	<a href="#">ARKodeGetNumPrecSolves()</a>
No. of linear iterations	<a href="#">ARKodeGetNumLinIters()</a>
No. of linear convergence failures	<a href="#">ARKodeGetNumLinConvFails()</a>
No. of Jacobian-vector setup evaluations	<a href="#">ARKodeGetNumJTSetupEvals()</a>
No. of Jacobian-vector product evaluations	<a href="#">ARKodeGetNumJtimesEvals()</a>
No. of $f_j$ calls for finite diff. $J$ or $Jv$ evals.	<a href="#">ARKodeGetNumLinRhsEvals()</a>
Last return from a linear solver function	<a href="#">ARKodeGetLastLinFlag()</a>
Name of constant associated with a return flag	<a href="#">ARKodeGetLinReturnFlagName()</a>
Size of real and integer mass matrix solver workspaces	<a href="#">ARKodeGetMassWorkSpace()</a>
No. of mass matrix solver setups (incl. $M$ evals.)	<a href="#">ARKodeGetNumMassSetups()</a>
No. of mass matrix multiply setups	<a href="#">ARKodeGetNumMassMultSetups()</a>
No. of mass matrix multiplies	<a href="#">ARKodeGetNumMassMult()</a>
No. of mass matrix solves	<a href="#">ARKodeGetNumMassSolves()</a>
No. of mass matrix preconditioner evaluations	<a href="#">ARKodeGetNumMassPrecEvals()</a>
No. of mass matrix preconditioner solves	<a href="#">ARKodeGetNumMassPrecSolves()</a>
No. of mass matrix linear iterations	<a href="#">ARKodeGetNumMassIters()</a>
No. of mass matrix solver convergence failures	<a href="#">ARKodeGetNumMassConvFails()</a>
No. of mass-matrix-vector setup evaluations	<a href="#">ARKodeGetNumMTSetups()</a>
Last return from a mass matrix solver function	<a href="#">ARKodeGetLastMassFlag()</a>

int **ARKodeGetJac**(void \*arkode\_mem, *SUNMatrix* \*J)

Returns the internally stored copy of the Jacobian matrix of the ODE implicit right-hand side function.

#### Parameters

- **arkode\_mem** – the ARKODE memory structure.
- **J** – the Jacobian matrix.

#### Return values

- **ARKLS\_SUCCESS** – the output value has been successfully set.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

#### Warning

This function is provided for debugging purposes and the values in the returned matrix should not be altered.

Added in version 6.1.0.

int **ARKodeGetJacTime**(void \*arkode\_mem, *sunrealtype* \*t\_J)

Returns the time at which the internally stored copy of the Jacobian matrix of the ODE implicit right-hand side function was evaluated.

#### Parameters

- **arkode\_mem** – the ARKODE memory structure.
- **t\_J** – the time at which the Jacobian was evaluated.

#### Return values

- **ARKLS\_SUCCESS** – the output value has been successfully set.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

int **ARKodeGetJacNumSteps**(void \*arkode\_mem, long int \*nst\_J)

Returns the value of the internal step counter at which the internally stored copy of the Jacobian matrix of the ODE implicit right-hand side function was evaluated.

#### Parameters

- **arkode\_mem** – the ARKODE memory structure.
- **nst\_J** – the value of the internal step counter at which the Jacobian was evaluated.

#### Return values

- **ARKLS\_SUCCESS** – the output value has been successfully set.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Added in version 6.1.0.

int **ARKodeGetLinWorkSpace**(void \*arkode\_mem, long int \*lenrwLS, long int \*leniwLS)

Returns the real and integer workspace used by the ARKLS linear solver interface.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **lenrwLS** – the number of `sunrealtype` values in the ARKLS workspace.
- **leniwLS** – the number of integer values in the ARKLS workspace.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the `SUNLinearSolver` object attached to it. The template Jacobian matrix allocated by the user outside of ARKLS is not included in this report.

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

Added in version 6.1.0.

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

int **ARKodeGetNumJacEvals**(void \*arkode\_mem, long int \*njevals)

Returns the number of Jacobian evaluations.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **njevals** – number of Jacobian evaluations.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumPrecEvals**(void \*arkode\_mem, long int \*npevals)

Returns the total number of preconditioner evaluations, i.e. the number of calls made to *psetup* with *jok* = *SUNFALSE* and that returned *\*jcurPtr* = *SUNTRUE*.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **npevals** – the current number of calls to *psetup*.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was *NULL*.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was *NULL*.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumPrecSolves**(void \*arkode\_mem, long int \*npsolves)

Returns the number of calls made to the preconditioner solve function, *psolve*.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **npsolves** – the number of calls to *psolve*.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was *NULL*.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was *NULL*.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumLinIters**(void \*arkode\_mem, long int \*nlinits)

Returns the cumulative number of linear iterations.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nliters** – the current number of linear iterations.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumLinConvFails**(void \*arkode\_mem, long int \*nlcfails)

Returns the cumulative number of linear convergence failures.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nlcfails** – the current number of linear convergence failures.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumJTSetupEvals**(void \*arkode\_mem, long int \*njtsetup)

Returns the cumulative number of calls made to the user-supplied Jacobian-vector setup function, *jtsetup*.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **njtsetup** – the current number of calls to *jtsetup*.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumJtimesEvals**(void \*arkode\_mem, long int \*njvevals)

Returns the cumulative number of calls made to the Jacobian-vector product function, *jtimes*.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **njvevals** – the current number of calls to *jtimes*.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumLinRhsEvals**(void \*arkode\_mem, long int \*nfevalsLS)

Returns the number of calls to the user-supplied implicit right-hand side function  $f^I$  for finite difference Jacobian or Jacobian-vector product approximation.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nfevalsLS** – the number of calls to the user implicit right-hand side function.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.

- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

The value `nfevalsLS` is incremented only if the default internal difference quotient function is used.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetLastLinFlag**(void \*arkode\_mem, long int \*lsflag)

Returns the last return value from an ARKLS routine.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **lsflag** – the value of the last return flag from an ARKLS function.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – linear solvers are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

If the ARKLS setup function failed when using the `SUNLINSOL_DENSE` or `SUNLINSOL_BAND` modules, then the value of `lsflag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix. For all other failures, `lsflag` is negative.

Otherwise, if the ARKLS setup function failed (`ARKodeEvolve()` returned `ARK_LSETUP_FAIL`), then `lsflag` will be `SUNLS_PSET_FAIL_UNREC`, `SUNLS_ASET_FAIL_UNREC` or `SUN_ERR_EXT_FAIL`.

If the ARKLS solve function failed (`ARKodeEvolve()` returned `ARK_LSOLVE_FAIL`), then `lsflag` contains the error return flag from the `SUNLinearSolver` object, which will be one of: `SUN_ERR_ARG_CORRUPT`, indicating that the `SUNLinearSolver` memory is NULL; `SUNLS_ATIMES_NULL`, indicating that a matrix-free iterative solver was provided, but is missing a routine for the matrix-vector product approximation; `SUNLS_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the  $Jv$  function; `SUNLS_PSOLVE_NULL`, indicating that an iterative linear solver was configured to use preconditioning, but no preconditioner solve routine was provided; `SUNLS_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function failed unrecoverably; `SUNLS_GS_FAIL`, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only); `SUNLS_QRSOL_FAIL`, indicating that the matrix  $R$  was found to be singular



during the QR solve phase (SPGMR and SPFGMR only); or *SUN\_ERR\_EXT\_FAIL*, indicating an unrecoverable failure in an external iterative linear solver package.

Added in version 6.1.0.

char \***ARKodeGetLinReturnFlagName**(long int lsflag)

Returns the name of the ARKLS constant corresponding to *lsflag*.

#### Parameters

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

#### Returns

The return value is a string containing the name of the corresponding constant. If using the *SUNLINSOL\_DENSE* or *SUNLINSOL\_BAND* modules, then if  $1 \leq lsflag \leq n$  (LU factorization failed), this routine returns “NONE”.

#### Warning

The user is responsible for freeing the returned string.

#### Note

This is only compatible with time-stepping modules that support implicit algebraic solvers.

Added in version 6.1.0.

int **ARKodeGetMassWorkSpace**(void \*arkode\_mem, long int \*lenrwMLS, long int \*leniwMLS)

Returns the real and integer workspace used by the ARKLS mass matrix linear solver interface.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **lenrwMLS** – the number of *sunrealtype* values in the ARKLS mass solver workspace.
- **leniwMLS** – the number of integer values in the ARKLS mass solver workspace.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the *SUNLinearSolver* object attached to it. The template mass matrix allocated by the user outside of ARKLS is not included in this report.

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

Added in version 6.1.0.

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

int **ARKodeGetNumMassSetups**(void \*arkode\_mem, long int \*nmsetups)

Returns the number of calls made to the ARKLS mass matrix solver ‘setup’ routine; these include all calls to the user-supplied mass-matrix constructor function.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmsetups** – number of calls to the mass matrix solver setup routine.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMassMultSetups**(void \*arkode\_mem, long int \*nmvsetups)

Returns the number of calls made to the ARKLS mass matrix ‘matvec setup’ (matrix-based solvers) routine.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmvsetups** – number of calls to the mass matrix matrix-times-vector setup routine.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMassMult**(void \*arkode\_mem, long int \*nmmults)

Returns the number of calls made to the ARKLS mass matrix ‘matvec’ routine (matrix-based solvers) or the user-supplied *mtimes* routine (matrix-free solvers).

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmmults** – number of calls to the mass matrix solver matrix-times-vector routine.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMassSolves**(void \*arkode\_mem, long int \*nmsolves)

Returns the number of calls made to the ARKLS mass matrix solver ‘solve’ routine.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmsolves** – number of calls to the mass matrix solver solve routine.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMassPrecEvals**(void \*arkode\_mem, long int \*nmpevals)

Returns the total number of mass matrix preconditioner evaluations, i.e. the number of calls made to *psetup*.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmpevals** – the current number of calls to *psetup*.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMassPrecSolves**(void \*arkode\_mem, long int \*nmpsolves)

Returns the number of calls made to the mass matrix preconditioner solve function, *psolve*.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmpsolves** – the number of calls to *psolve*.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMassIters**(void \*arkode\_mem, long int \*nmiters)

Returns the cumulative number of mass matrix solver iterations.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmiters** – the current number of mass matrix solver linear iterations.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMassConvFails**(void \*arkode\_mem, long int \*nmcfails)

Returns the cumulative number of mass matrix solver convergence failures.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmcfails** – the current number of mass matrix solver convergence failures.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

#### Note

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetNumMTSetups**(void \*arkode\_mem, long int \*nmtsetup)

Returns the cumulative number of calls made to the user-supplied mass-matrix-vector product setup function, *mtsetup*.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nmtsetup** – the current number of calls to *mtsetup*.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKODE, or when ARKODE is resized.

Added in version 6.1.0.

int **ARKodeGetLastMassFlag**(void \*arkode\_mem, long int \*mlsflag)

Returns the last return value from an ARKLS mass matrix interface routine.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **mlsflag** – the value of the last return flag from an ARKLS mass matrix solver interface function.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARK\_STEPPER\_UNSUPPORTED** – non-identity mass matrices are not supported by the current time-stepping module.

**Note**

This is only compatible with time-stepping modules that support non-identity mass matrices.

The values of *mlsflag* for each of the various solvers will match those described above for the function [\*ARKodeGetLastLinFlag\(\)\*](#).

Added in version 6.1.0.

### 5.3.10.5 General usability functions

The following optional routine may be called by a user to inquire about existing solver parameters. While this would not typically be called during the course of solving an initial value problem, it may be useful for users wishing to better understand ARKODE.

Optional routine	Function name
Output all ARKODE solver parameters	<a href="#"><i>ARKodeWriteParameters()</i></a>

int **ARKodeWriteParameters**(void \*arkode\_mem, FILE \*fp)

Outputs all ARKODE solver parameters to the provided file pointer.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **fp** – pointer to use for printing the solver parameters.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.

#### Note

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.

This routine will be called by [\*ARKodeSetOptions\(\)\*](#) when using the key “arkid.write\_parameters”.

Added in version 6.1.0.

### 5.3.11 ARKODE data preallocation function

Since the multi-stage structure of most ARKODE methods results in data requirements that depend on the number of stages, ARKODE generally defers allocation of stage-related internal data until the first call to [\*ARKodeEvolve\(\)\*](#). However, in some cases the user may wish to preallocate this data earlier, for example to measure the memory footprint before beginning a calculation, or to check for allocation errors at an earlier time. To request that that ARKODE preallocate all stage-related internal data before the first call to [\*ARKodeEvolve\(\)\*](#), the user may call the function [\*ARKodeInit\(\)\*](#).

int **ARKodeInit**(void \*arkode\_mem)

Optionally allocates internal data for the current ARKODE time-stepper module.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.

#### Return values

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – arkode\_mem was NULL.
- **ARK\_MEM\_FAIL** – a memory allocation failed.

**Warning**

This must be called **after** all other optional input routines have been called, and **before** the first call to [ARKodeEvolve\(\)](#). This routine should be called at most once per ARKODE memory block.

Added in version 6.7.0.

**5.3.12 ARKODE reset function**

To reset the ARKODE module to a particular state  $(t_R, y(t_R))$  for the continued solution of a problem, where a prior call to `*StepCreate` has been made, the user must call the function [ARKodeReset\(\)](#). Like the stepper-specific `*StepReInit` functions, this routine retains the current settings for all solver options and performs no memory allocations but, unlike `*StepReInit`, this routine performs only a *subset* of the input checking and initializations that are done in `*StepCreate`. In particular this routine retains all internal counter values and the step size/error history and does not reinitialize the linear and/or nonlinear solver but it does indicate that a linear solver setup is necessary in the next step. Like `*StepReInit`, a call to [ARKodeReset\(\)](#) will delete any previously-set *tstop* value specified via a call to [ARKodeSetStopTime\(\)](#). Following a successful call to [ARKodeReset\(\)](#), call [ARKodeEvolve\(\)](#) again to continue solving the problem. By default the next call to [ARKodeEvolve\(\)](#) will use the step size computed by ARKODE prior to calling [ARKodeReset\(\)](#). To set a different step size or have ARKODE estimate a new step size use [ARKodeSetInitStep\(\)](#).

One important use of the [ARKodeReset\(\)](#) function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to [ARKodeReset\(\)](#). To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **ARKodeReset** (void \*arkode\_mem, *sunrealtype* tR, *N\_Vector* yR)

Resets the current ARKODE time-stepper module state to the provided independent variable value and dependent variable vector.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **tR** – the value of the independent variable  $t$ .
- **yR** – the value of the dependent variable vector  $y(t_R)$ .

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_MEM\_FAIL** – a memory allocation failed.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

**Note**



By default the next call to `ARKodeEvolve()` will use the step size computed by ARKODE prior to calling `ARKodeReset()`. To set a different step size or have ARKODE estimate a new step size use `ARKodeSetInitStep()`.

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, `ARKodeReset()` also sends an error message to the error handler function.

#### Warning

Calling `ARKodeReset()` during forward integration of an IVP with checkpointing for adjoint sensitivity analysis is not supported.

Added in version 6.1.0.

### 5.3.13 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 a call to `*StepCreate`.

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 yR, sunrealtype hscale, sunrealtype tR, ARKVecResizeFn resize,
                void *resize_data)
```

Re-sizes ARKODE with a different state vector but with comparable dynamical time scale.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **yR** – the newly-sized state vector, holding the current dependent variable values  $y(t_R)$ .
- **hscale** – the desired time step scaling factor (i.e. the next step will be of size  $h*hscale$ ).
- **tR** – the current value of the independent variable  $t_R$  (this must be consistent with yR).
- **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 values

- **ARK\_SUCCESS** – the function exited successfully.

- **ARK\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARK\_NO\_MALLOC** – `arkode_mem` was not allocated.
- **ARK\_ILL\_INPUT** – an argument had an illegal value.

**Note**

If an error occurred, [`ARKodeResize\(\)`](#) also sends an error message to the error handler function.

If inequality constraint checking is enabled a call to [`ARKodeResize\(\)`](#) will disable constraint checking. A call to [`ARKodeSetConstraints\(\)`](#) is required to re-enable constraint checking.

**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 ARKLS interface should then be deleted and recreated by attaching the updated `SUNLinearSolver` (and possibly `SUNMatrix`) object(s) through calls to [`ARKodeSetLinearSolver\(\)`](#), and [`ARKodeSetMassLinearSolver\(\)`](#).

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 possibly [`ARKodeResVtolerance\(\)`](#) if applicable.

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.

**Example codes:**

- `examples/arkode/C_serial/ark_heat1D_adapt.c`

Added in version 6.1.0.

### 5.3.14 Using an ARKODE solver as an MRISolver “inner” solver

When using an integrator from ARKODE as the inner (fast) integrator with MRISolver, the utility function [`ARKodeCreateMRISolverInnerStepper\(\)`](#) should be used to wrap the ARKODE memory block as an [`MRISolverInnerStepper`](#).

```
int ARKodeCreateMRISolverInnerStepper(void *inner_arkode_mem, MRISolverInnerStepper *stepper)
```

Wraps an ARKODE integrator as an [`MRISolverInnerStepper`](#) for use with MRISolver.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **stepper** – the [`MRISolverInnerStepper`](#) object to create.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.

- **ARK\_MEM\_FAIL** – a memory allocation failed.
- **ARK\_STEPPER\_UNSUPPORTED** – the time-stepping module does not currently support use as an inner stepper.

**Note**

Currently, ARKODE integrators based on ARKStep, ERKStep, and MRISStep support use as an MRISStep inner stepper.

**Example usage:**

```
/* fast (inner) and slow (outer) ARKODE objects */
void *inner_arkode_mem = NULL;
void *outer_arkode_mem = NULL;

/* MRISStepInnerStepper to wrap the inner (fast) object */
MRISStepInnerStepper stepper = NULL;

/* create an ARKODE object, setting fast (inner) right-hand side
   functions and the initial condition */
inner_arkode_mem = *StepCreate(...);

/* configure the inner integrator */
retval = ARKodeSet*(inner_arkode_mem, ...);

/* create MRISStepInnerStepper wrapper for the ARKODE integrator */
flag = ARKodeCreateMRISStepInnerStepper(inner_arkode_mem, &stepper);

/* create an MRISStep object, setting the slow (outer) right-hand side
   functions and the initial condition */
outer_arkode_mem = MRISStepCreate(fse, fsi, t0, y0, stepper, suncctx)
```

### 5.3.15 Using an ARKODE solver as a SUNStepper

The utility function `ARKodeCreateSUNStepper()` wraps an ARKODE memory block as a *SUNStepper*.

int **ARKodeCreateSUNStepper**(void \*inner\_arkode\_mem, *SUNStepper* \*stepper)

Wraps an ARKODE integrator as a *SUNStepper*.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **stepper** – the *SUNStepper* object.

**Return values**

- **ARK\_SUCCESS** – the function exited successfully.
- **ARK\_MEM\_FAIL** – a memory allocation failed.
- **ARK\_SUNSTEPPER\_ERR** – the *SUNStepper* initialization failed.

**Warning**

Currently, `stepper` will be equipped with an implementation for the `SUNStepper_SetForcing()` function only if `inner_arkode_mem` is an ARKStep, ERKStep, or MRISStep integrator.

Added in version 6.2.0.

## 5.4 User-supplied functions

The user-supplied functions for ARKODE consist of:

- at least one function *defining the ODE* (required),
- a function that *provides the error weight vector* (optional),
- a function that *provides the residual weight vector* (optional),
- a function that *handles explicit time step stability* (optional),
- a function that *updates the implicit stage prediction* (optional),
- a function that *defines auxiliary temporal root-finding problem(s) to solve* (optional),
- one or two functions that *provide Jacobian-related information* for the linear solver, if a component is treated implicitly and a Newton-based nonlinear iteration is chosen (optional),
- one or two functions that *define the preconditioner* for use in any of the Krylov iterative algorithms, if linear systems of equations are to be solved using an iterative method (optional),
- if the problem involves a non-identity mass matrix  $M \neq I$  with ARKStep:
  - 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 if an iterative mass matrix solver is chosen (optional),
- a function that *handles vector resizing operations*, if the underlying vector structure supports resizing (as opposed to deletion/recreation), and if the user plans to call `ARKodeResize()` (optional),
- MRISStep only: functions to be *called before and after each inner integration* to perform any communication or memory transfers of forcing data supplied by the outer integrator to the inner integrator, or state data supplied by the inner integrator to the outer integrator,
- if relaxation is enabled (optional), a function that *evaluates the conservative or dissipative function*  $\xi(y(t))$  (required) and a function to *evaluate its Jacobian*  $\xi'(y(t))$  (required),
- functions that can optionally be called *before each step attempt and after each successful step*,
- a function that can optionally be called *before right-hand side function evaluations*, and
- functions that can optionally be called *after each step or stage computation* within supported ARKODE time stepping modules.

### 5.4.1 ODE right-hand side

The user must supply at least one function of type [ARKRhsFn](#) to specify the IVP-defining right-hand side function(s) when creating the ARKODE time-stepping module:

```
typedef int (*ARKRhsFn)(sunrealtype 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  $y$ .

**Parameters:**

- **t** – the current value of the independent variable.
- **y** – the current value of the dependent variable vector.
- **ydot** – the output vector that forms [a portion of] the ODE RHS  $f(t, y)$ .
- **user\_data** – the *user\_data* pointer that was passed to [ARKodeSetUserData\(\)](#).

**Returns:**

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).

**Note**

Allocation of memory for *ydot* is handled within ARKODE.

The vector *ydot* may be uninitialized on input; it is the user's responsibility to fill this entire vector with meaningful values.

A recoverable failure error return from the *ARKRhsFn* is typically used to flag a value of the dependent variable  $y$  that is "illegal" in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, ARKODE will attempt to recover (possibly repeating the nonlinear iteration, or reducing the step size in *ARKodeEvolve*) 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 time-stepping module completes a successful stage, in which case *ARKodeEvolve* returns *ARK\_UNREC\_RHSFUNC\_ERR*. Finally, when ARKODE is run in fixed-step mode, it may halt on a recoverable error flag that would normally have resulted in a stepsize reduction.

### 5.4.2 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 §2.10.

```
typedef int (*ARKEwtFn)(N_Vector y, N_Vector ewt, void *user_data)
```

This function computes the WRMS error weights for the vector  $y$ .

**Parameters:**

- **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\(\)](#) function

**Returns:**

An *ARKEwtFn* function must return 0 if it successfully set the error weights, and -1 otherwise.

**Note**

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.

### 5.4.3 Residual weight function

**Warning**

The functions in this section are specific to time-stepping modules that support non-identity mass matrices.

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 §2.10.

```
typedef int (*ARKRwtFn)(N_Vector y, N_Vector rwt, void *user_data)
```

This function computes the WRMS residual weights for the vector *y*.

**Parameters:**

- **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\(\)](#).

**Returns:**

An *ARKRwtFn* function must return 0 if it successfully set the residual weights, and -1 otherwise.

**Note**

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.

### 5.4.4 Time step adaptivity function

#### Warning

The function in this section is only used in now-deprecated functions in ARKStep and ERKStep, and will be removed in a future release.

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 such that the error estimate for the next time step remains below 1.

```
typedef int (*ARKAdaptFn)(N_Vector y, sunrealtype t, sunrealtype h1, sunrealtype h2, sunrealtype h3, sunrealtype e1, sunrealtype e2, sunrealtype e3, int q, int p, sunrealtype *hnew, void *user_data)
```

This function implements a time step adaptivity algorithm that chooses  $h$  to satisfy the error tolerances.

#### Parameters:

- **y** – the current value of the dependent variable vector.
- **t** – the current value of the independent variable.
- **h1** – the current step size,  $t_n - t_{n-1}$ .
- **h2** – the previous step size,  $t_{n-1} - t_{n-2}$ .
- **h3** – the step size  $t_{n-2} - t_{n-3}$ .
- **e1** – the error estimate from the current step,  $n$ .
- **e2** – the error estimate from the previous step,  $n - 1$ .
- **e3** – the error estimate from the step  $n - 2$ .
- **q** – the global order of accuracy for the method.
- **p** – the global order of accuracy for the embedded method.
- **hnew** – the output value of the next step size.
- **user\_data** – a pointer to user data, the same as the  $h\_data$  parameter that was passed to [ARKStepSetAdaptivityFn\(\)](#) or [ERKStepSetAdaptivityFn\(\)](#).

#### Returns:

An [ARKAdaptFn](#) function should return 0 if it successfully set the next step size, and a non-zero value otherwise.

Deprecated since version 5.7.0: Use the SUNAdaptController infrastructure instead (see §13.1).

### 5.4.5 Explicit stability function

#### Warning

The functions in this section are specific to time-stepping modules that support temporal adaptivity.

A user may supply a function to predict the maximum stable step size for an explicit portion of their IVP. 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 the explicit right-hand side function 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, sunrealtype t, sunrealtype *hstab, void *user_data)
```

This function predicts the maximum stable step size for the explicit portion of the ODE system.

**Parameters:**

- **y** – the current value of the dependent variable vector.
- **t** – the current value of the independent variable.
- **hstab** – the output value with the absolute value of the maximum stable step size.
- **user\_data** – a pointer to user data, the same as the *estab\_data* parameter that was passed to [ARKodeSetStabilityFn\(\)](#).

**Returns:**

An *ARKExpStabFn* function should return 0 if it successfully set the upcoming stable step size, and a non-zero value otherwise.

**Note**

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.

### 5.4.6 Implicit stage prediction function

A user may supply a function to update the prediction for each implicit stage solution. If supplied, this routine will be called *after* any existing ARKODE predictor algorithm completes, so that the predictor may be modified by the user as desired. In this scenario, a user may provide a function of type [ARKStagePredictFn](#) to provide this implicit predictor to ARKODE. This function takes as input the already-predicted implicit stage solution and the corresponding “time” for that prediction; it then updates the prediction vector as desired. If the user-supplied routine will construct a full prediction (and thus the ARKODE prediction is irrelevant), it is recommended that the user *not* call [ARKodeSetPredictorMethod\(\)](#), thereby leaving the default trivial predictor in place.

```
typedef int (*ARKStagePredictFn)(sunrealtype t, N_Vector zpred, void *user_data)
```

This function updates the prediction for the implicit stage solution.

**Parameters:**

- **t** – the current value of the independent variable containing the “time” corresponding to the predicted solution.
- **zpred** – the ARKODE-predicted stage solution on input, and the user-modified predicted stage solution on output.
- **user\_data** – a pointer to user data, the same as the *user\_data* parameter that was passed to [ARKodeSetUserData\(\)](#).

**Returns:**

An *ARKStagePredictFn* function should return 0 if it successfully set the upcoming stable step size, and a non-zero value otherwise.



**Note**

This may be useful if there are bound constraints on the solution, and these should be enforced prior to beginning the nonlinear or linear implicit solver algorithm.

This routine is incompatible with the “minimum correction predictor” – option 5 to the routine [ARKodeSetPredictorMethod\(\)](#). If both are selected, then ARKODE will override its built-in implicit predictor routine to instead use option 0 (trivial predictor).

### 5.4.7 Rootfinding function

If a rootfinding problem is to be solved during integration of the ODE system, the user must supply a function of type [ARKRootFn](#).

```
typedef int (*ARKRootFn)(sunrealtype t, N_Vector y, sunrealtype *gout, void *user_data)
```

This function implements a vector-valued function  $g(t, y)$  such that roots are sought for the components  $g_i(t, y)$ ,  $i = 0, \dots, nrtfn-1$ .

**Parameters:**

- **t** – the current value of the independent variable.
- **y** – the current value of the dependent variable vector.
- **gout** – the output array, of length *nrtfn*, with components  $g_i(t, y)$ .
- **user\_data** – a pointer to user data, the same as the *user\_data* parameter that was passed to the `SetUserData` function

**Returns:**

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*).

**Note**

Allocation of memory for *gout* is handled within ARKODE.

### 5.4.8 Jacobian construction

If a matrix-based linear solver module is used (i.e., a non-NULL `SUNMatrix` object was supplied to [ARKodeSetLinearSolver\(\)](#), the user may provide a function of type [ARKLsJacFn](#) to provide the Jacobian approximation or [ARKLsLinSysFn](#) to provide an approximation of the linear system  $\mathcal{A}(t, y) = M(t) - \gamma J(t, y)$ .

```
typedef int (*ARKLsJacFn)(sunrealtype t, N_Vector y, N_Vector fy, SUNMatrix Jac, void *user_data, N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
```

This function computes the Jacobian matrix  $J(t, y) = \frac{\partial f^I}{\partial y}(t, y)$  (or an approximation to it).

**Parameters:**

- **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 `ARKLsJacFn` as temporary storage or work space.

**Returns:**

An `ARKLsJacFn` function should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKODE will attempt to correct, while ARKLS sets *last\_flag* to `ARKLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, [ARKodeEvolve\(\)](#) returns `ARK_LSETUP_FAIL` and ARKLS sets *last\_flag* to `ARKLS_JACFUNC_UNRECVR`).

**Note**

Information regarding 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 §9 for details).

When using a linear solver of type `SUNLINEARSOLVER_DIRECT`, prior to calling the user-supplied Jacobian function, the Jacobian matrix  $J(t, y)$  is zeroed out, so only nonzero elements need to be loaded into *Jac*.

With the default Newton nonlinear solver, each call to the user's `ARKLsJacFn()` function is preceded by a call to the implicit `ARKRhsFn()` user function with the same  $(t, y)$  arguments. Thus, the Jacobian function can use any auxiliary data that is computed and saved during the evaluation of  $f^I(t, y)$ . In the case of a user-supplied or external nonlinear solver, this is also true if the nonlinear system function is evaluated prior to calling the linear solver setup function (see §11.1.4 for more information).

If the user's `ARKLsJacFn` function uses difference quotient approximations, then it may need to access quantities not in the argument list, including 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 `ARKSodeGet*` functions listed in §5.3.10. The unit roundoff can be accessed as `SUN_UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

**dense**  $J(t, y)$ : 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  $(t, y)$ . Utility routines and accessor macros for the `SUNMATRIX_DENSE` module are documented in §9.3.

**banded**  $J(t, y)$ : A user-supplied banded Jacobian function must load the band matrix *Jac* with the elements of the Jacobian  $J(t, y)$  at the point  $(t, y)$ . Utility routines and accessor macros for the `SUNMATRIX_BAND` module are documented in §9.6.

**sparse**  $J(t, y)$ : 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. Utility routines and accessor macros for the `SUNMATRIX_SPARSE` type are documented in §9.8.

```
typedef int (*ARKLsLinSysFn)(sunrealtype t, N_Vector y, N_Vector fy, SUNMatrix A, SUNMatrix M,
sunbooleantype jok, sunbooleantype *jcur, sunrealtype gamma, void *user_data, N_Vector tmp1, N_Vector tmp2,
N_Vector tmp3)
```

This function computes the linear system matrix  $\mathcal{A}(t, y) = M(t) - \gamma J(t, y)$  (or an approximation to it).

**Parameters:**

- **t** – the current value of the independent variable.
- **y** – the current value of the dependent variable vector, namely the predicted value of  $y(t)$ .
- **fy** – the current value of the vector  $f^I(t, y)$ .
- **A** – the output linear system matrix.
- **M** – the current mass matrix (this input is NULL if  $M = I$ ).
- **jok** – is an input flag indicating whether the Jacobian-related data needs to be updated. The *jok* argument provides for the reuse of Jacobian data. When *jok* = SUNFALSE, the Jacobian-related data should be recomputed from scratch. When *jok* = SUNTRUE the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of *gamma*). A call with *jok* = SUNTRUE can only occur after a call with *jok* = SUNFALSE.
- **jcur** – is a pointer to a flag which should be set to SUNTRUE if Jacobian data was recomputed, or set to SUNFALSE if Jacobian data was not recomputed, but saved data was still reused.
- **gamma** – the scalar  $\gamma$  appearing in the Newton system matrix  $\mathcal{A} = M(t) - \gamma J(t, y)$ .
- **user\_data** – a pointer to user data, the same as the *user\_data* parameter that was passed to [ARKodeSetUserData\(\)](#).
- **tmp1, tmp2, tmp3** – pointers to memory allocated to variables of type N\_Vector which can be used by an ARKLSLinSysFn as temporary storage or work space.

#### Returns:

An *ARKLSLinSysFn* function should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKODE will attempt to correct, while ARKLS sets *last\_flag* to *ARKLS\_JACFUNC\_RECVR*), or a negative value if it failed unrecoverably (in which case the integration is halted, [ARKodeEvolve\(\)](#) returns *ARK\_LSETUP\_FAIL* and ARKLS sets *last\_flag* to *ARKLS\_JACFUNC\_UNRECVR*).

### 5.4.9 Jacobian-vector product

When using a matrix-free linear solver module for the implicit stage solves (i.e., a NULL-valued SUNMATRIX argument was supplied to [ARKodeSetLinearSolver\(\)](#), the user may provide a function of type [ARKLSJacTimesVecFn](#) in the following form, to compute matrix-vector products  $Jv$ . If such a function is not supplied, the default is a difference quotient approximation to these products.

```
typedef int (*ARKLSJacTimesVecFn)(N_Vector v, N_Vector Jv, sunrealtype t, N_Vector y, N_Vector fy, void
*user_data, N_Vector tmp)
```

This function computes the product  $Jv$  where  $J(t, y) \approx \frac{\partial f^I}{\partial y}(t, y)$  (or an approximation to it).

#### Parameters:

- **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.

**Returns:**

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.

**Note**

If the user's `ARKLsJacTimesVecFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the `ARKodeGet*` functions listed in §5.3.10. The unit roundoff can be accessed as `SUN_UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

### 5.4.10 Jacobian-vector product setup

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

```
typedef int (*ARKLsJacTimesSetupFn)(sunrealtype 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.

**Parameters:**

- **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()`.

**Returns:**

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).

**Note**

Each call to the Jacobian-vector setup function is preceded by a call to the implicit `ARKRhsFn` user function with the same  $(t, y)$  arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side.

If the user's `ARKLsJacTimesSetupFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the `ARKodeGet*` functions listed in §5.3.10. The unit roundoff can be accessed as `SUN_UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

### 5.4.11 Preconditioner solve

If a user-supplied preconditioner is to be used with a SUNLinSol solver module, then the user must provide a function of type [ARKLsPrecSolveFn](#) to solve the linear system  $Pz = r$ , where  $P$  corresponds to either a left or right preconditioning matrix. Here  $P$  should approximate (at least crudely) the Newton matrix  $\mathcal{A}(t, y) = M(t) - \gamma J(t, y)$ , where  $M(t)$  is the mass matrix and  $J(t, y) = \frac{\partial f^I}{\partial y}(t, y)$ . If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate  $\mathcal{A}$ .

```
typedef int (*ARKLsPrecSolveFn)(sunrealtype t, N_Vector y, N_Vector fy, N_Vector r, N_Vector z, sunrealtype gamma, sunrealtype delta, int lr, void *user_data)
```

This function solves the preconditioner system  $Pz = r$ .

#### Parameters:

- **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  $\gamma$  appearing in the Newton matrix given by  $\mathcal{A} = M(t) - \gamma J(t, y)$ .
- **delta** – an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector  $Res = r - Pz$  of the system should be made to be less than *delta* in the weighted  $l_2$  norm, i.e.  $\left( \sum_{i=1}^n (Res_i * ewt_i)^2 \right)^{1/2} < \delta$ , where  $\delta = delta$ . To obtain the N\_Vector *ewt*, call [ARKodeGetErrorWeights\(\)](#).
- **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\(\)](#).

#### Returns:

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).

### 5.4.12 Preconditioner setup

If the user's preconditioner routine above requires that any data be preprocessed or evaluated, then these actions need to occur within a user-supplied function of type [ARKLsPrecSetupFn](#).

```
typedef int (*ARKLsPrecSetupFn)(sunrealtype t, N_Vector y, N_Vector fy, sunbooleantype jok, sunbooleantype *jcurPtr, sunrealtype gamma, void *user_data)
```

This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner.

#### Parameters:

- **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  $\gamma$  appearing in the Newton matrix given by  $\mathcal{A} = M(t) - \gamma J(t, y)$ .
- **user\_data** – a pointer to user data, the same as the *user\_data* parameter that was passed to `ARKodeSetUserData()`.

**Returns:**

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).

**Note**

The operations performed by this function might include forming a crude approximate Jacobian, and performing an LU factorization of the resulting approximation to  $\mathcal{A} = M(t) - \gamma J(t, y)$ .

With the default nonlinear solver (the native SUNDIALS Newton method), each call to the preconditioner setup function is preceded by a call to the implicit `ARKRhsFn` user function with the same  $(t, y)$  arguments. Thus, the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side. In the case of a user-supplied or external nonlinear solver, this is also true if the nonlinear system function is evaluated prior to calling the linear solver setup function (see §11.1.4 for more information).

This function is not called in advance of every call to the preconditioner solve function, but rather is called only as often as needed to achieve convergence in the Newton iteration.

If the user's `ARKLsPrecSetupFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their *user\_data*, and then use the `ARKodeGet*` functions listed in §5.3.10. The unit roundoff can be accessed as `SUN_UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

### 5.4.13 Mass matrix construction

For problems involving a non-identity mass matrix, if a matrix-based mass-matrix linear solver is used (i.e., a non-NULL `SUNMATRIX` was supplied to `ARKodeSetMassLinearSolver()`, the user must provide a function of type `ARKLsMassFn` to provide the mass matrix approximation.

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

This function computes the mass matrix  $M(t)$  (or an approximation to it).

**Parameters:**

- **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 `ARKLSMassFn` as temporary storage or work space.

**Returns:**

An `ARKLSMassFn` function should return 0 if successful, or a negative value if it failed unrecoverably (in which case the integration is halted, [ARKodeEvolve\(\)](#) returns `ARK_MASSSETUP_FAIL` and `ARKLS` sets *last\_flag* to `ARKLS_MASSFUNC_UNRECVR`).

**Note**

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 §9 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**  $M(t)$ : A user-supplied dense mass matrix function must load the  $N$  by  $N$  dense matrix  $M$  with an approximation to the mass matrix  $M(t)$ . Utility routines and accessor macros for the `SUNMATRIX_DENSE` module are documented in §9.3.

**banded**  $M(t)$ : A user-supplied banded mass matrix function must load the band matrix  $M$  with the elements of the mass matrix  $M(t)$ . Utility routines and accessor macros for the `SUNMATRIX_BAND` module are documented in §9.6.

**sparse**  $M(t)$ : 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. Utility routines and accessor macros for the `SUNMATRIX_SPARSE` type are documented in §9.8.

### 5.4.14 Mass matrix-vector product

For problems involving a non-identity mass matrix, if a matrix-free linear solver is to be used for mass-matrix linear systems (i.e., a `NULL`-valued `SUNMATRIX` argument was supplied to [ARKodeSetMassLinearSolver\(\)](#) in §5.2), the user *must* provide a function of type [ARKLSMassTimesVecFn](#) in the following form, to compute matrix-vector products  $M(t)v$ .

```
typedef int (*ARKLSMassTimesVecFn)(N_Vector v, N_Vector Mv, sunrealtype t, void *mtimes_data)
```

This function computes the product  $M(t)v$  (or an approximation to it).

**Parameters:**

- **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 [ARKodeSetMassTimes\(\)](#).

**Returns:**



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.

### 5.4.15 Mass matrix-vector product setup

For problems involving a non-identity mass matrix and a matrix-free linear solver, if the user's mass-matrix-times-vector routine requires that any mass matrix-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of type [ARKLSMassTimesSetupFn](#), defined as follows:

```
typedef int (*ARKLSMassTimesSetupFn)(sunrealtype t, void *mtimes_data)
```

This function preprocesses and/or evaluates any mass-matrix-related data needed by the mass-matrix-times-vector routine.

**Parameters:**

- **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 [ARKodeSetMassTimes\(\)](#).

**Returns:**

The value to be returned by the mass-matrix-vector setup function should be 0 if successful. Any other return value will result in an unrecoverable error of the ARKLS mass matrix solver interface, in which case the integration is halted.

### 5.4.16 Mass matrix preconditioner solve

For problems involving a non-identity mass matrix and an iterative linear solver, if a user-supplied preconditioner is to be used with a SUNLINEAR solver module for mass matrix linear systems, then the user must provide a function of type [ARKLSMassPrecSolveFn](#) to solve the linear system  $Pz = r$ , where  $P$  may be either a left or right preconditioning matrix. Here  $P$  should approximate (at least crudely) the mass matrix  $M(t)$ . If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate  $M(t)$ .

```
typedef int (*ARKLSMassPrecSolveFn)(sunrealtype t, N_Vector r, N_Vector z, sunrealtype delta, int lr, void *user_data)
```

This function solves the preconditioner system  $Pz = r$ .

**Parameters:**

- **t** – the current value of the independent variable.
- **r** – the right-hand side vector of the linear system.
- **z** – the computed output solution vector.
- **delta** – an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector  $Res = r - Pz$  of the system should be made to be less than *delta* in the weighted  $l_2$  norm, i.e.  $\left( \sum_{i=1}^n (Res_i * ewt_i)^2 \right)^{1/2} < \delta$ , where  $\delta = delta$ . To obtain the *N\_Vector* *ewt*, call [ARKodeGetErrorWeights\(\)](#).
- **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\(\)](#).



**Returns:**

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).

**5.4.17 Mass matrix preconditioner setup**

For problems involving a non-identity mass matrix and an iterative linear solver, if the user's mass matrix preconditioner above requires that any problem data be preprocessed or evaluated, then these actions need to occur within a user-supplied function of type [ARKLsMassPrecSetupFn](#).

```
typedef int (*ARKLsMassPrecSetupFn)(sunrealtype t, void *user_data)
```

This function preprocesses and/or evaluates mass-matrix-related data needed by the preconditioner.

**Parameters:**

- **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\(\)](#).

**Returns:**

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).

**Note**

The operations performed by this function might include forming a mass matrix and performing an incomplete factorization of the result. Although such operations would typically be performed only once at the beginning of a simulation, these may be required if the mass matrix can change as a function of time.

If both this function and a [ARKLsMassTimesSetupFn](#) are supplied, all calls to this function will be preceded by a call to the [ARKLsMassTimesSetupFn](#), so any setup performed there may be reused.

**5.4.18 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*.

**Parameters:**

- **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\(\)](#).

**Returns:**

An *ARKVecResizeFn* function should return 0 if it successfully resizes the vector *y*, and a non-zero value otherwise.

**Note**

If this function is not supplied, then ARKODE will instead destroy the vector *y* and clone a new vector *y* off of *ytemplate*.

### 5.4.19 Pre inner integrator communication function (MRISStep only)

The user may supply a function of type *MRISStepPreInnerFn* that will be called *before* each inner integration to perform any communication or memory transfers of forcing data supplied by the outer integrator to the inner integrator for the inner integration.

```
typedef int (*MRISStepPreInnerFn)(sunrealtype t, N_Vector *f, int num_vecs, void *user_data)
```

**Parameters:**

- **t** – the current value of the independent variable.
- **f** – an *N\_Vector* array of outer forcing vectors.
- **num\_vecs** – the number of vectors in the *N\_Vector* array.
- **user\_data** – the *user\_data* pointer that was passed to [ARKodeSetUserData\(\)](#).

**Returns:**

An *MRISStepPreInnerFn* function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred.

**Note**

In a heterogeneous computing environment if any data copies between the host and device vector data are necessary, this is where that should occur.

### 5.4.20 Post inner integrator communication function (MRISStep only)

The user may supply a function of type *MRISStepPostInnerFn* that will be called *after* each inner integration to perform any communication or memory transfers of state data supplied by the inner integrator to the outer integrator for the outer integration.

```
typedef int (*MRISStepPostInnerFn)(sunrealtype t, N_Vector y, void *user_data)
```

**Parameters:**

- **t** – the current value of the independent variable.
- **y** – the current value of the dependent variable vector.
- **user\_data** – the *user\_data* pointer that was passed to [ARKodeSetUserData\(\)](#).

**Returns:**

An *MRIStepPostInnerFn()* function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred.

#### Note

In a heterogeneous computing environment if any data copies between the host and device vector data are necessary, this is where that should occur.

### 5.4.21 Relaxation function

```
typedef int (*ARKRelaxFn)(N_Vector y, sunrealtype *r, void *user_data)
```

When applying relaxation, an *ARKRelaxFn()* function is required to compute the conservative or dissipative function  $\xi(y)$ .

#### Parameters:

- **y** – the current value of the dependent variable vector.
- **r** – the value of  $\xi(y)$ .
- **user\_data** – the user\_data pointer that was passed to *ARKodeSetUserData()*.

#### Returns:

An *ARKRelaxFn()* function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred. If a recoverable error occurs, the step size will be reduced and the step repeated.

### 5.4.22 Relaxation Jacobian function

```
typedef int (*ARKRelaxJacFn)(N_Vector y, N_Vector J, void *user_data);
```

When applying relaxation, an *ARKRelaxJacFn()* function is required to compute the Jacobian  $\xi'(y)$  of the *ARKRelaxFn()*  $\xi(y)$ .

#### Parameters:

- **y** – the current value of the dependent variable vector.
- **J** – the Jacobian vector  $\xi'(y)$ .
- **user\_data** – the user\_data pointer that was passed to *ARKodeSetUserData()*.

#### Returns:

An *ARKRelaxJacFn()* function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred. If a recoverable error occurs, the step size will be reduced and the step repeated.

### 5.4.23 Pre-Step, Post-Step, Pre-RHS, and Post-processing functions (ADVANCED)

The user may supply functions that will be called before each time step attempt, after each successful step, before their *ARKRhsFn* problem-defining functions are called, and after each internal step/stage computation. These user-supplied functions vary slightly in type, depending on their use case, as outlined below. Such functions are typically used for applications that compute auxiliary diagnostic data between time steps or stages, and store that data in their user\_data pointer or output to screen or disk. Alternately, some may be used to better prepare auxiliary data for an upcoming time

step or [ARKRhsFn](#) evaluation. **These should not be used to modify the active state data; if so then all theoretical guarantees of solution accuracy and stability will be lost.**

A user-provided [ARKPreStepFn](#) will be called before each internal time step attempt by ARKODE (see [ARKodeSetPreStepFn\(\)](#)).

```
typedef int (*ARKPreStepFn)(sunrealtype t, N_Vector y, long int step, int attempt, void *user_data)
```

A function to be called before each internal step attempt.

**Danger**

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

**Parameters:**

- **t** – the current value of the independent variable.
- **y** – the current value of the dependent variable vector that will be used as the initial condition for the upcoming step.
- **step** – the step index (starting at 0 for the first internal step since ARKODE was (re-)initialized).
- **attempt** – a counter indicating which attempt at the step is about to occur – 0 indicates that the previous step succeeded and this is the first attempt the step **step**, while a number greater than 0 indicates that the previous step attempt failed and this is a subsequent try at computing the step **step**.
- **user\_data** – the `user_data` pointer that was passed to [ARKodeSetUserData\(\)](#).

**Returns:**

An [ARKPreStepFn\(\)](#) function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred.

Added in version 6.7.0.

A user-provided [ARKPostStepFn](#) will be called following each *successful* internal time step by ARKODE (see [ARKodeSetPostStepFn\(\)](#)).

```
typedef int (*ARKPostStepFn)(sunrealtype t, N_Vector y, long int step, void *user_data)
```

A function to be called after each successful step attempt.

**Danger**

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

**Parameters:**

- **t** – the current value of the independent variable.
- **y** – the current value of the dependent variable vector that resulted from the successful time step.
- **step** – the step index (starting at 0 for the first internal step since ARKODE was (re-)initialized).
- **user\_data** – the `user_data` pointer that was passed to [ARKodeSetUserData\(\)](#).

**Returns:**

An [ARKPostStepFn\(\)](#) function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred.

Added in version 6.7.0.

A user-provided [ARKPreRhsFn](#) will be called just prior to any user-supplied [ARKRhsFn](#) (see [ARKodeSetPreRhsFn\(\)](#)). In the case of partitioned integration methods (e.g., ARKStep, MRISep), if multiple [ARKRhsFn](#) will be called with the same  $(t, y)$  argument, then the [ARKPreRhsFn](#) will be called only once just prior to the first [ARKRhsFn](#) that will be called with that  $(t, y)$  input.

```
typedef int (*ARKPreRhsFn)(sunrealtype t, N_Vector y, void *user_data)
```

A function to be called before right-hand side (RHS) evaluations.

#### Danger

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

#### Parameters:

- **t** – the current value of the independent variable that will be provided to the [ARKRhsFn](#).
- **y** – the current value of the dependent variable vector that will be provided to the [ARKRhsFn](#).
- **user\_data** – the user\_data pointer that was passed to [ARKodeSetUserData\(\)](#).

#### Returns:

An [ARKPreRhsFn\(\)](#) function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred.

Added in version 6.7.0.

A user-provided [ARKPostProcessFn](#) will be called either after each internal stage (see [ARKodeSetPostprocessStageFn\(\)](#)) or after each internal step attempt (see [ARKodeSetPostprocessStepFn\(\)](#)).

```
typedef int (*ARKPostProcessFn)(sunrealtype t, N_Vector y, void *user_data)
```

A function to postprocess step or stage data.

#### Danger

If the supplied function modifies any of the active state data, then all theoretical guarantees of solution accuracy and stability are lost.

#### Parameters:

- **t** – the current value of the independent variable.
- **y** – the current value of the dependent variable vector resulting from the step or stage.
- **user\_data** – the user\_data pointer that was passed to [ARKodeSetUserData\(\)](#).

#### Returns:

An [ARKPostProcessFn\(\)](#) function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred.

**Warning**

*ARKPostProcessFn()* functions are currently incompatible with discrete adjoint capabilities in ARKODE (*ARKodeSetAdjointCheckpointScheme()* and *ARKodeSetAdjointCheckpointIndex()*).

Added in version 6.7.0.

## 5.5 Relaxation Methods

This section describes user-callable functions for applying relaxation methods with ARKODE. For more information on relaxation Runge–Kutta methods see §2.18.

**Warning**

Relaxation support has not been evaluated with non-identity mass matrices. While this usage mode is supported, feedback from users who explore this combination would be appreciated.

### 5.5.1 Enabling or Disabling Relaxation

int **ARKodeSetRelaxFn**(void \*arkode\_mem, *ARKRelaxFn* rfn, *ARKRelaxJacFn* rjac)

Attaches the user supplied functions for evaluating the relaxation function (*rfn*) and its Jacobian (*rjac*).

Both *rfn* and *rjac* are required and an error will be returned if only one of the functions is NULL. If both *rfn* and *rjac* are NULL, relaxation is disabled.

With DIRK and IMEX-ARK methods or when a fixed mass matrix is present, applying relaxation requires allocating  $s$  additional state vectors (where  $s$  is the number of stages in the method).

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **rfn** – the user-defined function to compute the relaxation function  $\xi(y)$
- **rjac** – the user-defined function to compute the relaxation Jacobian  $\xi'(y)$

**Return values**

- **ARK\_SUCCESS** – the function exited successfully
- **ARK\_MEM\_NULL** – *arkode\_mem* was NULL
- **ARK\_ILL\_INPUT** – an invalid input combination was provided (see the output error message for more details)
- **ARK\_MEM\_FAIL** – a memory allocation failed

**Warning**

Applying relaxation requires using a method of at least second order with  $b_i^E \geq 0$  and  $b_i^I \geq 0$ . If these conditions are not satisfied, *ARKodeEvolve()* will return with an error during initialization.

**Note**

When combined with fixed time step sizes, ARKODE will attempt each step using the specified step size. If the step is successful, relaxation will be applied, effectively modifying the step size for the current step. If the step fails or applying relaxation fails, [ARKodeEvolve\(\)](#) will return with an error.

Added in version 6.1.0.

## 5.5.2 Optional Input Functions

This section describes optional input functions used to control applying relaxation.

int **ARKodeSetRelaxEtaFail**(void \*arkode\_mem, *sunrealtype* eta\_rf)

Sets the step size reduction factor applied after a failed relaxation application.

The default value is 0.25. Input values  $\leq 0$  or  $\geq 1$  will result in the default value being used.

### Parameters

- **arkode\_mem** – the ARKODE memory structure
- **eta\_rf** – the step size reduction factor

### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeSetRelaxLowerBound**(void \*arkode\_mem, *sunrealtype* lower)

Sets the smallest acceptable value for the relaxation parameter.

Values smaller than the lower bound will result in a failed relaxation application and the step will be repeated with a smaller step size (determined by [ARKodeSetRelaxEtaFail\(\)](#)).

The default value is 0.8. Input values  $\leq 0$  or  $\geq 1$  will result in the default value being used.

### Parameters

- **arkode\_mem** – the ARKODE memory structure
- **lower** – the relaxation parameter lower bound

### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeSetRelaxUpperBound**(void \*arkode\_mem, *sunrealtype* upper)

Sets the largest acceptable value for the relaxation parameter.

Values larger than the upper bound will result in a failed relaxation application and the step will be repeated with a smaller step size (determined by [ARKodeSetRelaxEtaFail\(\)](#)).

The default value is 1.2. Input values  $\leq 1$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **upper** – the relaxation parameter upper bound

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeSetRelaxMaxFails**(void \*arkode\_mem, int max\_fails)

Sets the maximum number of times applying relaxation can fail within a step attempt before the integration is halted with an error.

The default value is 10. Input values  $\leq 0$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **max\_fails** – the maximum number of failed relaxation applications allowed in a step

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeSetRelaxMaxIters**(void \*arkode\_mem, int max\_iters)

Sets the maximum number of nonlinear iterations allowed when solving for the relaxation parameter.

If the maximum number of iterations is reached before meeting the solve tolerance (determined by [ARKodeSetRelaxResTol\(\)](#) and [ARKodeSetRelaxTol\(\)](#)), the step will be repeated with a smaller step size (determined by [ARKodeSetRelaxEtaFail\(\)](#)).

The default value is 10. Input values  $\leq 0$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **max\_iters** – the maximum number of solver iterations allowed

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeSetRelaxSolver**(void \*arkode\_mem, [ARKRelaxSolver](#) solver)

Sets the nonlinear solver method used to compute the relaxation parameter.

The default value is `ARK_RELAX_NEWTON`.

**Parameters**



- **arkode\_mem** – the ARKODE memory structure
- **solver** – the nonlinear solver to use: `ARK_RELAX_BRENT` or `ARK_RELAX_NEWTON`

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL
- **ARK\_ILL\_INPUT** – an invalid solver option was provided

Added in version 6.1.0.

int **ARKodeSetRelaxResTol**(void \*arkode\_mem, *sunrealtype* res\_tol)

Sets the nonlinear solver residual tolerance to use when solving (2.63).

If the residual or iteration update tolerance (see [`ARKodeSetRelaxMaxIters\(\)`](#)) is not reached within the maximum number of iterations (determined by [`ARKodeSetRelaxMaxIters\(\)`](#)), the step will be repeated with a smaller step size (determined by [`ARKodeSetRelaxEtaFail\(\)`](#)).

The default value is  $4\epsilon$  where  $\epsilon$  is floating-point precision. Input values  $\leq 0$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **res\_tol** – the nonlinear solver residual tolerance to use

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeSetRelaxTol**(void \*arkode\_mem, *sunrealtype* rel\_tol, *sunrealtype* abs\_tol)

Sets the nonlinear solver relative and absolute tolerance on changes in  $r$  iterates when solving (2.63).

If the residual (see [`ARKodeSetRelaxResTol\(\)`](#)) or iterate update tolerance is not reached within the maximum number of iterations (determined by [`ARKodeSetRelaxMaxIters\(\)`](#)), the step will be repeated with a smaller step size (determined by [`ARKodeSetRelaxEtaFail\(\)`](#)).

The default relative and absolute tolerances are  $4\epsilon$  and  $10^{-14}$ , respectively, where  $\epsilon$  is floating-point precision. Input values  $\leq 0$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **rel\_tol** – the nonlinear solver relative solution tolerance to use
- **abs\_tol** – the nonlinear solver absolute solution tolerance to use

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

### 5.5.3 Optional Output Functions

This section describes optional output functions used to retrieve information about the performance of the relaxation method.

int **ARKodeGetNumRelaxFnEvals**(void \*arkode\_mem, long int \*r\_evals)

Get the number of times the user's relaxation function was evaluated.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **r\_evals** – the number of relaxation function evaluations

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeGetNumRelaxJacEvals**(void \*arkode\_mem, long int \*J\_evals)

Get the number of times the user's relaxation Jacobian was evaluated.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **J\_evals** – the number of relaxation Jacobian evaluations

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeGetNumRelaxFails**(void \*arkode\_mem, long int \*fails)

Get the total number of times applying relaxation failed.

The counter includes the sum of the number of nonlinear solver failures (see [\*ARKodeGetNumRelaxSolveFails\(\)\*](#)) and the number of failures due an unacceptable relaxation value (see [\*ARKodeSetRelaxLowerBound\(\)\*](#) and [\*ARKodeSetRelaxUpperBound\(\)\*](#)).

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **fails** – the total number of failed relaxation attempts

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeGetNumRelaxBoundFails**(void \*arkode\_mem, long int \*fails)

Get the number of times the relaxation parameter was deemed unacceptable.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **fails** – the number of failures due to an unacceptable relaxation parameter value

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeGetNumRelaxSolveFails**(void \*arkode\_mem, long int \*fails)

Get the number of times the relaxation parameter nonlinear solver failed.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **fails** – the number of relaxation nonlinear solver failures

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

int **ARKodeGetNumRelaxSolveIters**(void \*arkode\_mem, long int \*iters)

Get the number of relaxation parameter nonlinear solver iterations.

**Parameters**

- **arkode\_mem** – the ARKODE memory structure
- **iters** – the number of relaxation nonlinear solver iterations

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 6.1.0.

## 5.6 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 and threaded problems (ARKBANDPRE) and a band-block-diagonal preconditioner for parallel problems (ARKBBDPRE).

### 5.6.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with iterative SUNLINSOL modules in a serial or threaded setting. It requires that the problem be set up using either the NVECTOR\_SERIAL, NVECTOR\_OPENMP or NVECTOR\_PTHREADS module, due to data access patterns. It also currently requires that the problem involve an identity mass matrix, i.e.,  $M = I$ .

This module uses difference quotients of the ODE right-hand side function  $f^I$  to generate a band matrix of bandwidth  $ml + mu + 1$ , where the number of super-diagonals ( $mu$ , the upper half-bandwidth) and sub-diagonals ( $ml$ , the lower half-bandwidth) are specified by the user. This band matrix is used to 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.

#### 5.6.1.1 ARKBANDPRE usage

In order to use the ARKBANDPRE module, the user need not define any additional functions. In addition to the header files required for the integration of the ODE problem (see §5.1), 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 §5.2 are *italicized*.

1. *Initialize multi-threaded environment (if appropriate)*
2. *Create the SUNDIALS simulation context object.*
3. *Set problem dimensions*
4. *Set vector of initial values*
5. *Create ARKODE object*
6. *Specify integration tolerances*
7. Create iterative linear solver object

When creating the iterative linear solver object, specify the type of preconditioning (SUN\_PREC\_LEFT or SUN\_PREC\_RIGHT) to use.

8. *Set linear solver optional inputs*
9. *Attach linear solver module*
10. 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.

11. *Create nonlinear solver object*
12. *Attach nonlinear solver module*
13. *Set nonlinear solver optional inputs*
14. *Set optional inputs*

Note that the user should not call `ARKodeSetPreconditioner()` as it will overwrite the preconditioner setup and solve functions.

15. *Specify rootfinding problem*

16. *Advance solution in time*

17. *Get optional outputs*

Additional optional outputs associated with ARKBANDPRE are available by way of the two routines described below, [ARKBandPrecGetWorkspace\(\)](#) and [ARKBandPrecGetNumRhsEvals\(\)](#).

18. *Deallocate memory for solution vector*

19. *Free solver memory*

20. *Free linear solver memory*

21. *Free nonlinear solver memory*

### 5.6.1.2 ARKBANDPRE user-callable functions

The ARKBANDPRE preconditioner module is initialized and attached by calling the following function:

```
int ARKBandPrecInit(void *arkode_mem, sunindextype N, sunindextype mu, sunindextype ml)
```

Initializes the ARKBANDPRE preconditioner and allocates required (internal) memory for it.

#### Parameters

- **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 values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARKLS\_MEM\_FAIL** – a memory allocation request failed.

#### Note

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.

#### Parameters

- **arkode\_mem** – pointer to the ARKODE memory block.
- **lenrwLS** – the number of *sunrealtype* values in the ARKBANDPRE workspace.
- **leniwLS** – the number of integer values in the ARKBANDPRE workspace.

#### Return values

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_PMEM\_NULL** – the preconditioner memory was NULL.

**Note**

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 [ARKodeGetLinWorkspace\(\)](#).

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

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.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **nfevalsBP** – number of calls to  $f^I$ .

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_PMEM\_NULL** – the preconditioner memory was NULL.

**Note**

The counter *nfevalsBP* is distinct from the counter *nfevalsLS* returned by the corresponding function [ARKodeGetNumLinRhsEvals\(\)](#) and also from the number of evaluations returned by the time-stepping module (e.g., *nfi\_evals* returned by [ARKStepGetNumRhsEvals\(\)](#)). The total number of right-hand side function evaluations is the sum of all three of these counters.

## 5.6.2 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 be 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 [61], and is included in a software module within the ARKODE package. This preconditioning module works with the parallel vector module NVECTOR\_

PARALLEL and is usable with any of the Krylov iterative linear solvers through the ARKLS interface. It generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called ARKBBDPRE.

One way to envision these preconditioners is to think of the computational PDE domain as being subdivided into  $Q$  non-overlapping subdomains, where each subdomain is assigned to one of the  $Q$  MPI tasks used to solve the ODE system. The basic idea is to isolate the preconditioning so that it is local to each process, and also to use a (possibly cheaper) approximate right-hand side function for construction of this preconditioning matrix. This requires the definition of a new function  $g(t, y) \approx f^I(t, y)$  that will be used to construct the BBD preconditioner matrix. At present, we assume that the ODE be written in explicit form as

$$\dot{y} = f^E(t, y) + f^I(t, y),$$

where  $f^I$  corresponds to the ODE components to be treated implicitly, i.e. this preconditioning module does not support problems with non-identity mass matrices. The user may set  $g = f^I$ , if no less expensive approximation is desired.

Corresponding to the domain decomposition, there is a decomposition of the solution vector  $y$  into  $Q$  disjoint blocks  $y_q$ , and a decomposition of  $g$  into blocks  $g_q$ . The block  $g_q$  depends both on  $y_p$  and on components of blocks  $y_{q'}$  associated with neighboring subdomains (so-called ghost-cell data). If we let  $\bar{y}_q$  denote  $y_q$  augmented with those other components on which  $g_q$  depends, then we have

$$g(t, y) = [g_1(t, \bar{y}_1), g_2(t, \bar{y}_2), \dots, g_Q(t, \bar{y}_Q)]^T,$$

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

$$P = \begin{bmatrix} P_1 & & & \\ & P_2 & & \\ & & \ddots & \\ & & & P_Q \end{bmatrix}$$

where

$$P_q \approx I - \gamma J_q$$

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

$$P_q x_q = b_q, \quad q = 1, \dots, Q,$$

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.

### 5.6.2.1 ARKBBDPRE user-supplied functions

The ARKBBDPRE module calls two user-provided functions to construct  $P$ : a required function  $gloc$  (of type [ARKLocalFn\(\)](#)) which approximates the right-hand side function  $g(t, y) \approx f^I(t, y)$  and which is computed locally, and an optional function  $cfn$  (of type [ARKCommFn\(\)](#)) which performs all inter-process communication necessary to evaluate the approximate right-hand side  $g$ . These are in addition to the user-supplied right-hand side function  $f^I$ . Both functions take as input the same pointer  $user\_data$  that is passed by the user to [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, sunrealtype 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  $y$ .

**Param Nlocal**

the local vector length.

**Param t**

the value of the independent variable.

**Param y**

the value of the dependent variable vector on this process.

**Param glocal**

the output vector of  $g(t, y)$  on this process.

**Param user\_data**

a pointer to user data, the same as the  $user\_data$  parameter passed to [ARKodeSetUserData\(\)](#).

**Return**

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 [ARKodeEvolve\(\)](#) will return [ARK\\_LSETUP\\_FAIL](#)).

**Note**

This function should assume that all inter-process communication of data needed to calculate  $glocal$  has already been done, and that this data is accessible within user data.

The case where  $g$  is mathematically identical to  $f^I$  is allowed.

```
typedef int (*ARKCommFn)(sunindextype Nlocal, sunrealtype t, N_Vector y, void *user_data)
```

This  $cfn$  function performs all inter-process communication necessary for the execution of the  $gloc$  function above, using the input vector  $y$ .

**Param Nlocal**

the local vector length.

**Param t**

the value of the independent variable.

**Param y**

the value of the dependent variable vector on this process.

**Param user\_data**

a pointer to user data, the same as the  $user\_data$  parameter passed to [ARKodeSetUserData\(\)](#).

**Return**

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 `ARKodeEvolve()` will return `ARK_LSETUP_FAIL`).

#### Note

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).

### 5.6.2.2 ARKBBDPRE usage

In addition to the header files required for the integration of the ODE problem (see §5.1), 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 §5.2 are *italicized*.

1. *Initialize MPI*
2. *Create the SUNDIALS simulation context object*
3. *Set problem dimensions*
4. *Set vector of initial values*
5. *Create ARKODE object*
6. *Specify integration tolerances*
7. Create iterative linear solver object

When creating the iterative linear solver object, specify the type of preconditioning (`SUN_PREC_LEFT` or `SUN_PREC_RIGHT`) to use.

8. *Set linear solver optional inputs*
9. *Attach linear solver module*
10. Initialize the ARKBBDPRE preconditioner module

Specify the upper and lower half-bandwidths for computation `mudq` and `mldq`, the upper and lower half-bandwidths for storage `mukeep` and `mlkeep`, and call

```
ier = ARKBBDPrecInit(arkode_mem, Nlocal, mudq, mldq, mukeep, mlkeep, dqrely, gloc, cfn);
```

to allocate memory and initialize the internal preconditioner data. The last two arguments of `ARKBBDPrecInit()` are the two user-supplied functions of type `ARKLocalFn()` and `ARKCommFn()` described above, respectively.

11. *Create nonlinear solver object*
12. *Attach nonlinear solver module*
13. *Set nonlinear solver optional inputs*

14. *Set optional inputs*

Note that the user should not call `ARKodeSetPreconditioner()` as it will overwrite the preconditioner setup and solve functions.

15. *Specify rootfinding problem*16. *Advance solution in time*17. *Get optional outputs*

Additional optional outputs associated with ARKBBDPRE are available through the routines `ARKBBDPrecGetWorkSpace()` and `ARKBBDPrecGetNumGfnEvals()`.

18. *Deallocate memory for solution vector*19. *Free solver memory*20. *Free linear solver memory*21. *Free nonlinear solver memory*22. *Finalize MPI*

### 5.6.2.3 ARKBBDPRE user-callable functions

The ARKBBDPRE preconditioner module is initialized (or re-initialized) and attached to the integrator by calling the following functions:

```
int ARKBBDPrecInit(void *arkode_mem, sunindextype Nlocal, sunindextype mudq, sunindextype mldq,
                  sunindextype mukeep, sunindextype mlkeep, sunrealtype dqrely, ARKLocalFn gloc,
                  ARKCommFn cfn)
```

Initializes and allocates (internal) memory for the ARKBBDPRE preconditioner.

**Parameters**

- **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 inter-process communication required for the computation of  $g(t, y)$ .

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.

- **ARKLS\_ILL\_INPUT** – an input had an illegal value.
- **ARKLS\_MEM\_FAIL** – a memory allocation request failed.

**Note**

If one of the half-bandwidths *mudq* or *mldq* to be used in the difference quotient calculation of the approximate Jacobian is negative or exceeds the value *Nlocal*-1, it is replaced by 0 or *Nlocal*-1 accordingly.

The half-bandwidths *mudq* and *mldq* need not be the true half-bandwidths of the Jacobian of the local block of *g* when smaller values may provide a greater efficiency.

Also, the half-bandwidths *mukeep* and *mlkeep* of the retained banded approximate Jacobian block may be even smaller than *mudq* and *mldq*, to reduce storage and computational costs further.

For all four half-bandwidths, the values need not be the same on every processor.

The ARKBBDPRE module also provides a re-initialization function to allow solving a sequence of problems of the same size, with the same linear solver choice, provided there is no change in *Nlocal*, *mukeep*, or *mlkeep*. After solving one problem, and after calling `*StepReInit` 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 SUNLINSOL module, and/or one or more of the corresponding `ARKodeSet***` functions, must also be made (in the proper order).

int **ARKBBDPrecReInit**(void \*arkode\_mem, *sunindextype* mudq, *sunindextype* mldq, *sunrealtype* dqrely)

Re-initializes the ARKBBDPRE preconditioner module.

**Parameters**

- **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 values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – *arkode\_mem* was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_PMEM\_NULL** – the preconditioner memory was NULL.

**Note**

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.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **lenrwBBDP** – the number of `sunrealtype` values in the ARKBBDPRE workspace.
- **leniwBBDP** – the number of integer values in the ARKBBDPRE workspace.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_PMEM\_NULL** – the preconditioner memory was NULL.

**Note**

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 [ARKodeGetLinWorkspace\(\)](#).

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

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.

**Parameters**

- **arkode\_mem** – pointer to the ARKODE memory block.
- **ngevalsBBDP** – the number of calls made to the user-supplied *gloc* function.

**Return values**

- **ARKLS\_SUCCESS** – the function exited successfully.
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL.
- **ARKLS\_LMEM\_NULL** – the linear solver memory was NULL.
- **ARKLS\_PMEM\_NULL** – the preconditioner memory was 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 [§5.3.10.4](#)).

## 5.7 Using the ARKStep time-stepping module

This section is concerned with the use of the ARKStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. Usage of ARKStep follows that of the rest of ARKODE, and so in this section we primarily focus on those usage aspects that are specific to ARKStep.

### 5.7.1 ARKStep User-callable functions

This section describes the ARKStep-specific functions that may be called by the user to setup and then solve an IVP using the ARKStep time-stepping module. The large majority of these routines merely wrap *underlying ARKODE functions*, and are now deprecated – each of these are clearly marked. However, some of these user-callable functions are specific to ARKStep, as explained below.

As discussed in the main *ARKODE user-callable function introduction*, each of ARKODE’s time-stepping modules clarifies the categories of user-callable functions that it supports. ARKStep supports *all categories*:

- temporal adaptivity
- implicit nonlinear and/or linear solvers
- non-identity mass matrices
- relaxation Runge–Kutta methods

ARKStep also has forcing function support when converted to a *SUNStepper* or *MRISStepInnerStepper*. See *ARKodeCreateSUNStepper()* and *ARKStepCreateMRISStepInnerStepper()* for additional details.

#### 5.7.1.1 ARKStep initialization and deallocation functions

void **ARKStepCreate**(*ARKRhsFn* fe, *ARKRhsFn* fi, *sunrealtype* t0, *N\_Vector* y0, *SUNContext* sunctx)

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

**Arguments:**

- *fe* – the name of the C function (of type *ARKRhsFn()*) defining the explicit portion of the right-hand side function in  $M(t) y'(t) = f^E(t, y) + f^I(t, y)$ .
- *fi* – the name of the C function (of type *ARKRhsFn()*) defining the implicit portion of the right-hand side function in  $M(t) y'(t) = f^E(t, y) + f^I(t, y)$ .
- *t0* – the initial value of  $t$ .
- *y0* – the initial condition vector  $y(t_0)$ .
- *sunctx* – the *SUNContext* object (see §4.2)

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

void **ARKStepFree**(void \*\*arkode\_mem)

This function frees the problem memory *arkode\_mem* created by *ARKStepCreate()*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:** None

Deprecated since version 6.1.0: Use *ARKodeFree()* instead.

#### 5.7.1.2 ARKStep tolerance specification functions

int **ARKStepSStolerances**(void \*arkode\_mem, *sunrealtype* reltol, *sunrealtype* abstol)

This function specifies scalar relative and absolute tolerances.

**Arguments:**

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

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NO\_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. a negative tolerance).

Deprecated since version 6.1.0: Use [\*ARKodeSStolerances\(\)\*](#) instead.

int **ARKStepSVtolerances**(void \*arkode\_mem, *sunrealtype* reltol, *N\_Vector* abstol)

This function specifies a scalar relative tolerance and a vector absolute tolerance (a potentially different absolute tolerance for each vector component).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – vector containing the absolute tolerances for each solution component.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NO\_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. a negative tolerance).

Deprecated since version 6.1.0: Use [\*ARKodeSVtolerances\(\)\*](#) instead.

int **ARKStepWftolerances**(void \*arkode\_mem, *ARKEwtFn* efun)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *efun* – the name of the function (of type [\*ARKEwtFn\(\)\*](#)) that implements the error weight vector computation.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NO\_MALLOC* if the ARKStep memory was not allocated by the time-stepping module

Deprecated since version 6.1.0: Use [\*ARKodeWftolerances\(\)\*](#) instead.

int **ARKStepResStolerance**(void \*arkode\_mem, *sunrealtype* rabstol)

This function specifies a scalar absolute residual tolerance.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *rabstol* – scalar absolute residual tolerance.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NO\_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. a negative tolerance).

Deprecated since version 6.1.0: Use [ARKodeResStolerance\(\)](#) instead.

int **ARKStepResVtolerance**(void \*arkode\_mem, *N\_Vector* rabstol)

This function specifies a vector of absolute residual tolerances.

**Arguments:**

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

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NO\_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. a negative tolerance).

Deprecated since version 6.1.0: Use [ARKodeResVtolerance\(\)](#) instead.

int **ARKStepResFtolerance**(void \*arkode\_mem, *ARKRwtFn* rfun)

This function specifies a user-supplied function *rfun* to compute the residual weight vector *rwt*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *rfun* – the name of the function (of type [ARKRwtFn\(\)](#)) that implements the residual weight vector computation.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NO\_MALLOC* if the ARKStep memory was not allocated by the time-stepping module

Deprecated since version 6.1.0: Use [ARKodeResFtolerance\(\)](#) instead.

### 5.7.1.3 Linear solver interface functions

int **ARKStepSetLinearSolver**(void \*arkode\_mem, *SUNLinearSolver* LS, *SUNMatrix* J)

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

**Arguments:**

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

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_MEM\_FAIL* if there was a memory allocation failure
- *ARKLS\_ILL\_INPUT* if ARKLS is incompatible with the provided *LS* or *J* input objects, or the current *N\_Vector* module.

**Notes:**

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

If *LS* is a matrix-based linear solver, then the template Jacobian matrix *J* will be used in the solve process, so if additional storage is required within the *SUNMatrix* object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular *SUNMATRIX* type in the §9 for further information).

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

Deprecated since version 6.1.0: Use *ARKodeSetLinearSolver()* instead.

### 5.7.1.4 Mass matrix solver specification functions

int **ARKStepSetMassLinearSolver**(void \*arkode\_mem, *SUNLinearSolver* LS, *SUNMatrix* M, *sunbooleantype* time\_dep)

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

**Arguments:**

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

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL



- *ARKLS\_MEM\_FAIL* if there was a memory allocation failure
- *ARKLS\_ILL\_INPUT* if ARKLS is incompatible with the provided *LS* or *M* input objects, or the current *N\_Vector* module.

**Notes:**

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

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

If called with *time\_dep* set to *SUNFALSE*, then the mass matrix is only computed and factored once (or when either *ARKStepReInit()* or *ARKStepResize()* are called), with the results reused throughout the entire ARKStep simulation.

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

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

Deprecated since version 6.1.0: Use *ARKodeSetMassLinearSolver()* instead.

**5.7.1.5 Nonlinear solver interface functions**

int **ARKStepSetNonlinearSolver**(void \*arkode\_mem, *SUNNonlinearSolver* NLS)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *NLS* – the *SUNNonlinearSolver* object to use.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_MEM\_FAIL* if there was a memory allocation failure
- *ARK\_ILL\_INPUT* if ARKStep is incompatible with the provided *NLS* input object.

**Notes:**

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

Deprecated since version 6.1.0: Use *ARKodeSetNonlinearSolver()* instead.

**5.7.1.6 Rootfinding initialization function**

int **ARKStepRootInit**(void \*arkode\_mem, int nrtn, *ARKRootFn* g)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

- *nrtfn* – number of functions  $g_i$ , an integer  $\geq 0$ .
- *g* – name of user-supplied function, of type [ARKRootFn\(\)](#), defining the functions  $g_i$  whose roots are sought.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_MEM\_FAIL* if there was a memory allocation failure
- *ARK\_ILL\_INPUT* if *nrtfn* is greater than zero but *g* = NULL.

**Notes:**

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

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

Deprecated since version 6.1.0: Use [ARKodeRootInit\(\)](#) instead.

### 5.7.1.7 ARKStep solver function

int [ARKStepEvolve](#)(void \*arkode\_mem, *sunrealtype* tout, *N\_Vector* yout, *sunrealtype* \*tret, int itask)

Integrates the ODE over an interval in *t*.

**Arguments:**

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

The *ARK\_NORMAL* option causes the solver to take internal steps until it has just overtaken a user-specified output time, *tout*, in the direction of integration, i.e.  $t_{n-1} < tout \leq t_n$  for forward integration, or  $t_n \leq tout < t_{n-1}$  for backward integration. It will then compute an approximation to the solution  $y(tout)$  by interpolation (as described in §2.2).

The *ARK\_ONE\_STEP* option tells the solver to only take a single internal step,  $y_{n-1} \rightarrow y_n$ , and return the solution at that point,  $y_n$ , in the vector *yout*.

**Return value:**

- *ARK\_SUCCESS* if successful.
- *ARK\_ROOT\_RETURN* if [ARKStepEvolve\(\)](#) succeeded, and found one or more roots. If the number of root functions, *nrtfn*, is greater than 1, call [ARKStepGetRootInfo\(\)](#) to see which  $g_i$  were found to have a root at (\*tret).
- *ARK\_TSTOP\_RETURN* if [ARKStepEvolve\(\)](#) succeeded and returned at *tstop*.
- *ARK\_MEM\_NULL* if the *arkode\_mem* argument was NULL.
- *ARK\_NO\_MALLOC* if *arkode\_mem* was not allocated.

- **ARK\_ILL\_INPUT** if one of the inputs to [ARKStepEvolve\(\)](#) is illegal, or some other input to the solver was either illegal or missing. Details will be provided in the error message. Typical causes of this failure:
  - (a) A component of the error weight vector became zero during internal time-stepping.
  - (b) The linear solver initialization function (called by the user after calling [ARKStepCreate\(\)](#)) failed to set the linear solver-specific *lsolve* field in *arkode\_mem*.
  - (c) A root of one of the root functions was found both at a point *t* and also very near *t*.
  - (d) The initial condition violates the inequality constraints.
- **ARK\_TOO\_MUCH\_WORK** if the solver took *mxstep* internal steps but could not reach *tout*. The default value for *mxstep* is *MXSTEP\_DEFAULT* = 500.
- **ARK\_TOO\_MUCH\_ACC** if the solver could not satisfy the accuracy demanded by the user for some internal step.
- **ARK\_ERR\_FAILURE** if error test failures occurred either too many times (*ark\_maxnef*) during one internal time step or occurred with  $|h| = h_{min}$ .
- **ARK\_CONV\_FAILURE** if either convergence test failures occurred too many times (*ark\_maxncf*) during one internal time step or occurred with  $|h| = h_{min}$ .
- **ARK\_LINIT\_FAIL** if the linear solver's initialization function failed.
- **ARK\_LSETUP\_FAIL** if the linear solver's setup routine failed in an unrecoverable manner.
- **ARK\_LSOLVE\_FAIL** if the linear solver's solve routine failed in an unrecoverable manner.
- **ARK\_MASSINIT\_FAIL** if the mass matrix solver's initialization function failed.
- **ARK\_MASSSETUP\_FAIL** if the mass matrix solver's setup routine failed.
- **ARK\_MASSSOLVE\_FAIL** if the mass matrix solver's solve routine failed.
- **ARK\_VECTOROP\_ERR** a vector operation error occurred.

#### Notes:

The input vector *yout* can use the same memory as the vector *y0* of initial conditions that was passed to [ARKStepCreate\(\)](#).

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

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

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

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

Deprecated since version 6.1.0: Use [ARKodeEvolve\(\)](#) instead.

### 5.7.1.8 Optional input functions

#### Optional inputs for ARKStep

int **ARKStepSetDefaults**(void \*arkode\_mem)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Does not change the *user\_data* pointer or any parameters within the specified time-stepping module.

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

Deprecated since version 6.1.0: Use [ARKodeSetDefaults\(\)](#) instead.

int **ARKStepSetInterpolantType**(void \*arkode\_mem, int itype)

Deprecated since version 6.1.0: This function is now a wrapper to [ARKodeSetInterpolantType\(\)](#), see the documentation for that function instead.

int **ARKStepSetInterpolantDegree**(void \*arkode\_mem, int degree)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *degree* – requested polynomial degree.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory or interpolation module are NULL
- *ARK\_INTERP\_FAIL* if this is called after [ARKStepEvolve\(\)](#)
- *ARK\_ILL\_INPUT* if an argument had an illegal value or the interpolation module has already been initialized

**Notes:**

Allowed values are between 0 and 5.

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

If a user calls both this routine and [ARKStepSetInterpolantType\(\)](#), then [ARKStepSetInterpolantType\(\)](#) must be called first.

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

Changed in version 5.5.1: When  $q = 1$ , a linear interpolant is the default to ensure values obtained by the integrator are returned at the ends of the time interval.

Deprecated since version 6.1.0: Use [ARKodeSetInterpolantDegree\(\)](#) instead.

int **ARKStepSetDenseOrder**(void \*arkode\_mem, int dord)

Deprecated since version 5.2.0: Use [ARKodeSetInterpolantDegree\(\)](#) instead.

int **ARKStepSetDiagnostics**(void \*arkode\_mem, FILE \*diagfp)

Specifies the file pointer for a diagnostics file where all ARKStep step adaptivity and solver information is written.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *diagfp* – pointer to the diagnostics output file.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had 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.

Deprecated since version 5.2.0: Use [SUNLogger\\_SetInfoFilename\(\)](#) instead.

int **ARKStepSetFixedStep**(void \*arkode\_mem, *sunrealtype* hfixed)

Disables time step adaptivity within ARKStep, and specifies the fixed time step size to use for the following internal step(s).

**Arguments:**

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

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

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

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

When using [ARKStepSetFixedStep\(\)](#), any values provided to the functions [ARKStepSetInitStep\(\)](#), [ARKStepSetAdaptivityFn\(\)](#), [ARKStepSetMaxErrTestFails\(\)](#), [ARKStepSetAdaptivityMethod\(\)](#), [ARKStepSetCFLFraction\(\)](#), [ARKStepSetErrorBias\(\)](#), [ARKStepSetFixedStepBounds\(\)](#), [ARKStepSetMaxCFailGrowth\(\)](#), [ARKStepSetMaxEFailGrowth\(\)](#), [ARKStepSetMaxFirstGrowth\(\)](#), [ARKStepSetMaxGrowth\(\)](#), [ARKStepSetMinReduction\(\)](#), [ARKStepSetSafetyFac-](#)

`tor()`, `ARKStepSetSmallNumEFails()`, `ARKStepSetStabilityFn()`, and `ARKStepSetAdaptController()` will be ignored, since temporal adaptivity is disabled.

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

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

Deprecated since version 6.1.0: Use `ARKodeSetFixedStep()` instead.

int **ARKStepSetInitStep**(void \*arkode\_mem, *sunrealtype* hin)

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

**Arguments:**

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

**Return value:**

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

**Notes:**

Pass 0.0 to use the default value.

By default, ARKStep estimates the initial step size to be  $h = \sqrt{\frac{2}{\|\ddot{y}\|}}$ , where  $\ddot{y}$  is estimate of the second derivative of the solution at  $t_0$ .

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

Deprecated since version 6.1.0: Use `ARKodeSetInitStep()` instead.

int **ARKStepSetMaxHnilWarns**(void \*arkode\_mem, int mxhnil)

Specifies the maximum number of messages issued by the solver to warn that  $t + h = t$  on the next internal step, before ARKStep will instead return with an error.

**Arguments:**

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

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ARKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had 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.

Deprecated since version 6.1.0: Use `ARKodeSetMaxHnilWarns()` instead.

int **ARKStepSetMaxNumSteps**(void \*arkode\_mem, long int mxsteps)

Specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time, before ARKStep will return with an error.

**Arguments:**

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

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

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

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

Deprecated since version 6.1.0: Use [ARKodeSetMaxNumSteps\(\)](#) instead.

int **ARKStepSetMaxStep**(void \*arkode\_mem, *sunrealtype* hmax)

Specifies the upper bound on the magnitude of the time step size.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *hmax* – maximum absolute value of the time step size ( $\geq 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Pass *hmax*  $\leq 0.0$  to set the default value of  $\infty$ .

Deprecated since version 6.1.0: Use [ARKodeSetMaxStep\(\)](#) instead.

int **ARKStepSetMinStep**(void \*arkode\_mem, *sunrealtype* hmin)

Specifies the lower bound on the magnitude of the time step size.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *hmin* – minimum absolute value of the time step size ( $\geq 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

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

Deprecated since version 6.1.0: Use [ARKodeSetMinStep\(\)](#) instead.

int **ARKStepSetStopTime**(void \*arkode\_mem, *sunrealtype* tstop)

Specifies the value of the independent variable  $t$  past which the solution is not to proceed.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *tstop* – stopping time for the integrator.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default is that no stop time is imposed.

Once the integrator returns at a stop time, any future testing for *tstop* is disabled (and can be re-enabled only through a new call to [ARKStepSetStopTime\(\)](#)).

A stop time not reached before a call to [ARKStepReInit\(\)](#) or [ARKStepReset\(\)](#) will remain active but can be disabled by calling [ARKStepClearStopTime\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetStopTime\(\)](#) instead.

int **ARKStepSetInterpolateStopTime**(void \*arkode\_mem, *sunbooleantype* interp)

Specifies that the output solution should be interpolated when the current  $t$  equals the specified *tstop* (instead of merely copying the internal solution  $y_n$ ).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *interp* – flag indicating to use interpolation (1) or copy (0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetInterpolateStopTime\(\)](#) instead.

int **ARKStepClearStopTime**(void \*arkode\_mem)

Disables the stop time set with [ARKStepSetStopTime\(\)](#).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL

**Notes:**

The stop time can be re-enabled through a new call to [ARKStepSetStopTime\(\)](#).

Added in version 5.5.1.

Deprecated since version 6.1.0: Use [ARKodeClearStopTime\(\)](#) instead.



int **ARKStepSetUserData**(void \*arkode\_mem, void \*user\_data)

Specifies the user data block *user\_data* and attaches it to the main ARKStep memory block.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *user\_data* – pointer to the user data.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had 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 preconditioner functions, the call to this function must be made *before* any calls to [ARKStepSetLinearSolver\(\)](#) and/or [ARKStepSetMassLinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetUserData\(\)](#) instead.

int **ARKStepSetMaxErrTestFails**(void \*arkode\_mem, int maxnef)

Specifies the maximum number of error test failures permitted in attempting one step, before returning with an error.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *maxnef* – maximum allowed number of error test failures ( $> 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default value is 7; set  $maxnef \leq 0$  to specify this default.

Deprecated since version 6.1.0: Use [ARKodeSetMaxErrTestFails\(\)](#) instead.

int **ARKStepSetOptimalParams**(void \*arkode\_mem)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

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

Deprecated since version 6.1.0: Adjust solver parameters individually instead. For reference, this routine sets the following non-default parameters:

- Explicit methods:
  - *SUNAdaptController\_PI()* with *SUNAdaptController\_SetErrorBias()* of 1.2 and *SUNAdaptController\_SetParams\_PI()* of  $k_1 = 0.8$  and  $k_2 = -0.31$
  - *ARKodeSetSafetyFactor()* of 0.99
  - *ARKodeSetMaxGrowth()* of 25.0
  - *ARKodeSetMaxEFailGrowth()* of 0.3
- Implicit methods:
  - Order 3:
    - \* *SUNAdaptController\_I()* with *SUNAdaptController\_SetErrorBias()* of 1.9
    - \* *ARKodeSetSafetyFactor()* of 0.957
    - \* *ARKodeSetMaxGrowth()* of 17.6
    - \* *ARKodeSetMaxEFailGrowth()* of 0.45
    - \* *ARKodeSetNonlinConvCoef()* of 0.22
    - \* *ARKodeSetNonlinCRDown()* of 0.17
    - \* *ARKodeSetNonlinRDiv()* of 2.3
    - \* *ARKodeSetDeltaGammaMax()* of 0.19
  - Order 4:
    - \* *SUNAdaptController\_PID()* with *SUNAdaptController\_SetErrorBias()* of 1.2 and *SUNAdaptController\_SetParams\_PID()* of  $k_1 = 0.535$ ,  $k_2 = -0.209$ , and  $k_3 = 0.148$
    - \* *ARKodeSetSafetyFactor()* of 0.988
    - \* *ARKodeSetMaxGrowth()* of 31.5
    - \* *ARKodeSetMaxEFailGrowth()* of 0.33
    - \* *ARKodeSetNonlinConvCoef()* of 0.24
    - \* *ARKodeSetNonlinCRDown()* of 0.26
    - \* *ARKodeSetNonlinRDiv()* of 2.3
    - \* *ARKodeSetDeltaGammaMax()* of 0.16
    - \* *ARKodeSetLSetupFrequency()* of 31
  - Order 5:
    - \* *SUNAdaptController\_PID()* with *SUNAdaptController\_SetErrorBias()* of 3.3 and *SUNAdaptController\_SetParams\_PID()* of  $k_1 = 0.56$ ,  $k_2 = -0.338$ , and  $k_3 = 0.14$
    - \* *ARKodeSetSafetyFactor()* of 0.937
    - \* *ARKodeSetMaxGrowth()* of 22.0

- \* `ARKodeSetMaxEFailGrowth()` of 0.44
- \* `ARKodeSetNonlinConvCoef()` of 0.25
- \* `ARKodeSetNonlinCRDown()` of 0.4
- \* `ARKodeSetNonlinRDiv()` of 2.3
- \* `ARKodeSetDeltaGammaMax()` of 0.32
- \* `ARKodeSetLSetupFrequency()` of 31
- ImEx methods:
  - Order 2:
    - \* `ARKodeSetNonlinConvCoef()` of 0.001
    - \* `ARKodeSetMaxNonlinIters()` of 5
  - Order 3:
    - \* `SUNAdaptController_PID()` with `SUNAdaptController_SetErrorBias()` of 1.42 and `SUNAdaptController_SetParams_PID()` of  $k_1 = 0.54$ ,  $k_2 = -0.36$ , and  $k_3 = 0.14$
    - \* `ARKodeSetSafetyFactor()` of 0.965
    - \* `ARKodeSetMaxGrowth()` of 28.7
    - \* `ARKodeSetMaxEFailGrowth()` of 0.46
    - \* `ARKodeSetNonlinConvCoef()` of 0.22
    - \* `ARKodeSetNonlinCRDown()` of 0.17
    - \* `ARKodeSetNonlinRDiv()` of 2.3
    - \* `ARKodeSetDeltaGammaMax()` of 0.19
    - \* `ARKodeSetLSetupFrequency()` of 60
  - Order 4:
    - \* `SUNAdaptController_PID()` with `SUNAdaptController_SetErrorBias()` of 1.35 and `SUNAdaptController_SetParams_PID()` of  $k_1 = 0.543$ ,  $k_2 = -0.297$ , and  $k_3 = 0.14$
    - \* `ARKodeSetSafetyFactor()` of 0.97
    - \* `ARKodeSetMaxGrowth()` of 25.0
    - \* `ARKodeSetMaxEFailGrowth()` of 0.47
    - \* `ARKodeSetNonlinConvCoef()` of 0.24
    - \* `ARKodeSetNonlinCRDown()` of 0.26
    - \* `ARKodeSetNonlinRDiv()` of 2.3
    - \* `ARKodeSetDeltaGammaMax()` of 0.16
    - \* `ARKodeSetLSetupFrequency()` of 31
  - Order 5:
    - \* `SUNAdaptController_PI()` with `SUNAdaptController_SetErrorBias()` of 1.15 and `SUNAdaptController_SetParams_PI()` of  $k_1 = 0.8$  and  $k_2 = -0.35$
    - \* `ARKodeSetSafetyFactor()` of 0.993
    - \* `ARKodeSetMaxGrowth()` of 28.5

- \* [ARKodeSetMaxEFailGrowth\(\)](#) of 0.3
- \* [ARKodeSetNonlinConvCoef\(\)](#) of 0.25
- \* [ARKodeSetNonlinCRDown\(\)](#) of 0.4
- \* [ARKodeSetNonlinRDiv\(\)](#) of 2.3
- \* [ARKodeSetDeltaGammaMax\(\)](#) of 0.32
- \* [ARKodeSetLSetupFrequency\(\)](#) of 31

int **ARKStepSetConstraints**(void \*arkode\_mem, *N\_Vector* constraints)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *constraints* – vector of constraint flags. Each component specifies the type of solution constraint:

$$\text{constraints}[i] = \begin{cases} 0.0 & \Rightarrow & \text{no constraint is imposed on } y_i, \\ 1.0 & \Rightarrow & y_i \geq 0, \\ -1.0 & \Rightarrow & y_i \leq 0, \\ 2.0 & \Rightarrow & y_i > 0, \\ -2.0 & \Rightarrow & y_i < 0. \end{cases}$$

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if the constraints vector contains illegal values

**Notes:**

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

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

Since constraint-handling is performed through cutting time steps that would violate the constraints, it is possible that this feature will cause some problems to fail due to an inability to enforce constraints even at the minimum time step size. Additionally, the features [ARKStepSetConstraints\(\)](#) and [ARKStepSetFixedStep\(\)](#) are incompatible, and should not be used simultaneously.

Deprecated since version 6.1.0: Use [ARKodeSetConstraints\(\)](#) instead.

int **ARKStepSetMaxNumConstrFails**(void \*arkode\_mem, int maxfails)

Specifies the maximum number of constraint failures in a step before ARKStep will return with an error.

**Arguments:**

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

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL

**Notes:**

Passing *maxfails*  $\leq 0$  results in ARKStep using the default value (10).

Deprecated since version 6.1.0: Use [ARKodeSetMaxNumConstrFails\(\)](#) instead.

**Optional inputs for IVP method selection**

Optional input	Function name	Default
Set integrator method order	<a href="#">ARKStepSetOrder()</a>	4
Specify implicit/explicit problem	<a href="#">ARKStepSetImEx()</a>	SUNTRUE
Specify explicit problem	<a href="#">ARKStepSetExplicit()</a>	SUNFALSE
Specify implicit problem	<a href="#">ARKStepSetImplicit()</a>	SUNFALSE
Set additive RK tables	<a href="#">ARKStepSetTables()</a>	internal
Set additive RK tables via their numbers	<a href="#">ARKStepSetTableNum()</a>	internal
Set additive RK tables via their names	<a href="#">ARKStepSetTableName()</a>	internal

int **ARKStepSetOrder**(void \*arkode\_mem, int ord)

Specifies the order of accuracy for the ARK/DIRK/ERK integration method.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *ord* – requested order of accuracy.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

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

Since *ord* affects the memory requirements for the internal ARKStep memory block, it cannot be changed after the first call to [ARKStepEvolve\(\)](#), unless [ARKStepReInit\(\)](#) is called.

Deprecated since version 6.1.0: Use [ARKodeSetOrder\(\)](#) instead.

int **ARKStepSetImEx**(void \*arkode\_mem)

Specifies that both the implicit and explicit portions of problem are enabled, and to use an additive Runge–Kutta method.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This is automatically deduced when neither of the function pointers  $f_e$  or  $f_i$  passed to [ARKStepCreate\(\)](#) are NULL, but may be set directly by the user if desired.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.imex”.

int **ARKStepSetExplicit**(void \*arkode\_mem)

Specifies that the implicit portion of problem is disabled, and to use an explicit RK method.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This is automatically deduced when the function pointer  $f_i$  passed to [ARKStepCreate\(\)](#) is NULL, but may be set directly by the user if desired.

If the problem is posed in explicit form, i.e.  $\dot{y} = f(t, y)$ , then we recommend that the ERKStep time-stepper module be used instead.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.explicit”.

int **ARKStepSetImplicit**(void \*arkode\_mem)

Specifies that the explicit portion of problem is disabled, and to use a diagonally implicit RK method.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This is automatically deduced when the function pointer  $f_e$  passed to [ARKStepCreate\(\)](#) is NULL, but may be set directly by the user if desired.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.implicit”.

int **ARKStepSetTables**(void \*arkode\_mem, int q, int p, [ARKodeButcherTable](#) Bi, [ARKodeButcherTable](#) Be)

Specifies a customized Butcher table (or pair) for the ERK, DIRK, or ARK method.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- $q$  – global order of accuracy for the ARK method.
- $p$  – global order of accuracy for the embedded ARK method.
- $Bi$  – the Butcher table for the implicit RK method.
- $Be$  – the Butcher table for the explicit RK method.

**Return value:**

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

**Notes:**

For a description of the `ARKodeButcherTable` type and related functions for creating Butcher tables, see §6.

To set an explicit table, *Bi* must be NULL. This automatically calls `ARKStepSetExplicit()`. However, if the problem is posed in explicit form, i.e.  $\dot{y} = f(t, y)$ , then we recommend that the ERKStep time-stepper module be used instead of ARKStep.

To set an implicit table, *Be* must be NULL. This automatically calls `ARKStepSetImplicit()`.

If both *Bi* and *Be* are provided, this routine automatically calls `ARKStepSetImEx()`.

When only one table is provided (i.e., *Bi* or *Be* is NULL) then the input values of *q* and *p* are ignored and the global order of the method and embedding (if applicable) are obtained from the Butcher table structures. If both *Bi* and *Be* are non-NULL (e.g. an ImEx method is provided) then the input values of *q* and *p* are used as the order of the ARK method may be less than the orders of the individual tables. No error checking is performed to ensure that either *p* or *q* correctly describe the coefficients that were input.

Error checking is subsequently performed at ARKStep initialization to ensure that *Bi* and *Be* (if non-NULL) specify DIRK and ERK methods, respectively. Specifically, the *A* member of *Bi* must be lower triangular with at least one nonzero value on the diagonal, and the *A* member of *Be* must be strictly lower triangular. When both *Bi* and *Be* are non-NULL, they must agree on the number of internal stages, i.e., the *stages* members of both structures must match.

If the inputs *Bi* or *Be* do not contain an embedding (when the corresponding explicit or implicit table is non-NULL), the user *must* call `ARKStepSetFixedStep()` to enable fixed-step mode and set the desired time step size.

**Warning:**

This should not be used with `ARKodeSetOrder()`.

int `ARKStepSetTableNum`(void \*arkode\_mem, `ARKODE_DIRKTableID` itable, `ARKODE_ERKTableID` etable)

Indicates to use specific built-in Butcher tables for the ERK, DIRK or ARK method.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *itable* – index of the DIRK Butcher table.
- *etable* – index of the ERK Butcher table.

**Return value:**

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

**Notes:**

The allowable values for both the *itable* and *etable* arguments corresponding to built-in tables may be found in §19.

To choose an explicit table, set *itable* to a negative value. This automatically calls `ARKStepSetExplicit()`. However, if the problem is posed in explicit form, i.e.  $\dot{y} = f(t, y)$ , then we recommend that the ERKStep time-stepper module be used instead of ARKStep.

To select an implicit table, set *etable* to a negative value. This automatically calls [ARKStepSetImplicit\(\)](#).

If both *itable* and *etable* are non-negative, then these should match an existing implicit/explicit pair, listed in §19.3. This automatically calls [ARKStepSetImEx\(\)](#).

In all cases, error-checking is performed to ensure that the tables exist.

**Warning:**

This should not be used with [ARKodeSetOrder\(\)](#).

int **ARKStepSetTableName**(void \*arkode\_mem, const char \*itable, const char \*etable)

Indicates to use specific built-in Butcher tables for the ERK, DIRK or ARK method.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *itable* – name of the DIRK Butcher table.
- *etable* – name of the ERK Butcher table.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The allowable values for both the *itable* and *etable* arguments corresponding to built-in tables may be found in §19. This function is case sensitive.

To choose an explicit table, set *itable* to "ARKODE\_DIRK\_NONE". This automatically calls [ARKStepSetExplicit\(\)](#). However, if the problem is posed in explicit form, i.e.  $\dot{y} = f(t, y)$ , then we recommend that the ERKStep time-stepper module be used instead of ARKStep.

To select an implicit table, set *etable* to "ARKODE\_ERK\_NONE". This automatically calls [ARKStepSetImplicit\(\)](#).

If both *itable* and *etable* are not none, then these should match an existing implicit/explicit pair, listed in §19.3. This automatically calls [ARKStepSetImEx\(\)](#).

In all cases, error-checking is performed to ensure that the tables exist.

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key "arkid.table\_names".

**Warning:**

This should not be used with [ARKodeSetOrder\(\)](#).

### Optional inputs for time step adaptivity

int **ARKStepSetAdaptController**(void \*arkode\_mem, [SUNAdaptController](#) C)

Sets a user-supplied time-step controller object.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *C* – user-supplied time adaptivity controller. If NULL then the I controller will be created (see §13.2).

**Return value:**



- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ARKStep memory is NULL
- `ARK_MEM_FAIL` if `C` was NULL and the I controller could not be allocated.

Added in version 5.7.0.

Deprecated since version 6.1.0: Use `ARKodeSetAdaptController()` instead.

Changed in version 6.3.0: The default controller was changed from PID to I.

int **ARKStepSetAdaptivityFn**(void \*arkode\_mem, *ARKAdaptFn* hfun, void \*h\_data)

Sets a user-supplied time-step adaptivity function.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *hfun* – name of user-supplied adaptivity function.
- *h\_data* – pointer to user data passed to *hfun* every time it is called.

**Return value:**

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

**Notes:**

This function should focus on accuracy-based time step estimation; for stability based time steps the function `ARKStepSetStabilityFn()` should be used instead.

Deprecated since version 5.7.0: Use the SUNAdaptController infrastructure instead (see §13.1).

int **ARKStepSetAdaptivityMethod**(void \*arkode\_mem, int imethod, int ideofault, int pq, *sunrealtype* \*adapt\_params)

Specifies the method (and associated parameters) used for time step adaptivity.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *imethod* – accuracy-based adaptivity method choice ( $0 \leq imethod \leq 5$ ): 0 is PID, 1 is PI, 2 is I, 3 is explicit Gustafsson, 4 is implicit Gustafsson, and 5 is the ImEx Gustafsson.
- *ideofault* – 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), the method order of accuracy *q* (1), or the minimum of the two (any input not equal to 0 or 1) within the adaptivity algorithm. *p* is the default.
- *adapt\_params*[0] –  $k_1$  parameter within accuracy-based adaptivity algorithms.
- *adapt\_params*[1] –  $k_2$  parameter within accuracy-based adaptivity algorithms.
- *adapt\_params*[2] –  $k_3$  parameter within accuracy-based adaptivity algorithms.

**Return value:**

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

**Notes:**

If custom parameters are supplied, they will be checked for validity against published stability intervals. If other parameter values are desired, it is recommended to instead provide a custom function through a call to [ARKStepSetAdaptivityFn\(\)](#).

Changed in version 5.7.0: Prior to version 5.7.0, any nonzero value for  $pq$  would result in use of the embedding order of accuracy.

Deprecated since version 5.7.0: Use the SUNAdaptController infrastructure instead (see §13.1).

int **ARKStepSetAdaptivityAdjustment**(void \*arkode\_mem, int adjust)

Called by a user to adjust the method order supplied to the temporal adaptivity controller. For example, if the user expects order reduction due to problem stiffness, they may request that the controller assume a reduced order of accuracy for the method by specifying a value  $adjust < 0$ .

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *adjust* – adjustment factor (default is 0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This should be called prior to calling [ARKStepEvolve\(\)](#), and can only be reset following a call to [ARKStepReInit\(\)](#).

Added in version 5.7.0.

Deprecated since version 6.1.0: Use [ARKodeSetAdaptivityAdjustment\(\)](#) instead.

Changed in version 6.3.0: The default value was changed from -1 to 0

int **ARKStepSetCFLFraction**(void \*arkode\_mem, *sunrealtype* cfl\_frac)

Specifies the fraction of the estimated explicitly stable step to use.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *cfl\_frac* – maximum allowed fraction of explicitly stable step (default is 0.5).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetCFLFraction\(\)](#) instead.

int **ARKStepSetErrorBias**(void \*arkode\_mem, *sunrealtype* bias)

Specifies the bias to be applied to the error estimates within accuracy-based adaptivity strategies.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

- *bias* – bias applied to error in accuracy-based time step estimation (default is 1.0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value below 1.0 will imply a reset to the default value.

If both this and one of [ARKStepSetAdaptivityMethod\(\)](#) or [ARKStepSetAdaptController\(\)](#) will be called, then this routine must be called *second*.

Deprecated since version 5.7.0: Use the SUNAdaptController infrastructure instead (see §13.1).

Changed in version 6.3.0: The default value was changed from 1.5 to 1.0

int **ARKStepSetFixedStepBounds**(void \*arkode\_mem, *sunrealtype* lb, *sunrealtype* ub)

Specifies the step growth interval in which the step size will remain unchanged.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *lb* – lower bound on window to leave step size fixed (default is 1.0).
- *ub* – upper bound on window to leave step size fixed (default is 1.0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any interval *not* containing 1.0 will imply a reset to the default values.

Deprecated since version 6.1.0: Use [ARKodeSetFixedStepBounds\(\)](#) instead.

Changed in version 6.3.0: The default upper bound was changed from 1.5 to 1.0

int **ARKStepSetMaxCFailGrowth**(void \*arkode\_mem, *sunrealtype* etacf)

Specifies the maximum step size growth factor upon an algebraic solver convergence failure on a stage solve within a step,  $\eta_{cf}$  from §2.15.3.1.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *etacf* – time step reduction factor on a nonlinear solver convergence failure (default is 0.25).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value outside the interval (0, 1] will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetMaxCFailGrowth\(\)](#) instead.

int **ARKStepSetMaxEFailGrowth**(void \*arkode\_mem, *sunrealtype* etamxf)

Specifies the maximum step size growth factor upon multiple successive accuracy-based error failures in the solver.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *etamxf* – time step reduction factor on multiple error fails (default is 0.3).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value outside the interval  $(0, 1]$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [\*ARKodeSetMaxEFailGrowth\(\)\*](#) instead.

int **ARKStepSetMaxFirstGrowth**(void \*arkode\_mem, *sunrealtype* etamx1)

Specifies the maximum allowed growth factor in step size following the very first integration step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *etamx1* – maximum allowed growth factor after the first time step (default is 10000.0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value  $\leq 1.0$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [\*ARKodeSetMaxFirstGrowth\(\)\*](#) instead.

int **ARKStepSetMaxGrowth**(void \*arkode\_mem, *sunrealtype* mx\_growth)

Specifies the maximum allowed growth factor in step size between consecutive steps in the integration process.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *mx\_growth* – maximum allowed growth factor between consecutive time steps (default is 20.0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value  $\leq 1.0$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [\*ARKodeSetMaxGrowth\(\)\*](#) instead.

int **ARKStepSetMinReduction**(void \*arkode\_mem, *sunrealtype* eta\_min)

Specifies the minimum allowed reduction factor in step size between step attempts, resulting from a temporal error failure in the integration process.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *eta\_min* – minimum allowed reduction factor in time step after an error test failure (default is 0.1).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value outside the interval (0, 1) will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetMinReduction\(\)](#) instead.

int **ARKStepSetSafetyFactor**(void \*arkode\_mem, *sunrealtype* safety)

Specifies the safety factor to be applied to the accuracy-based estimated step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *safety* – safety factor applied to accuracy-based time step (default is 0.9).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value  $\leq 0$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetSafetyFactor\(\)](#) instead.

Changed in version 6.3.0: The default default was changed from 0.96 to 0.9. The maximum value is now exactly 1.0 rather than strictly less than 1.0.

int **ARKStepSetSmallNumEFails**(void \*arkode\_mem, int small\_nef)

Specifies the threshold for “multiple” successive error failures before the *etamxf* parameter from [ARKStepSetMaxEFailGrowth\(\)](#) is applied.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *small\_nef* – bound to determine ‘multiple’ for *etamxf* (default is 2).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value  $\leq 0$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetSmallNumEFails\(\)](#) instead.

int **ARKStepSetStabilityFn**(void \*arkode\_mem, [ARKExpStabFn](#) EStab, void \*estab\_data)

Sets the problem-dependent function to estimate a stable time step size for the explicit portion of the ODE system.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *EStab* – name of user-supplied stability function.
- *estab\_data* – pointer to user data passed to *EStab* every time it is called.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had 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  $f^E(t, y)$  contains stiff terms.

Deprecated since version 6.1.0: Use [ARKodeSetStabilityFn\(\)](#) instead.

### Optional inputs for implicit stage solves

int **ARKStepSetLinear**(void \*arkode\_mem, int timedepend)

Specifies that the implicit portion of the problem is linear.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *timedepend* – flag denoting whether the Jacobian of  $f^I(t, y)$  is time-dependent (1) or not (0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Tightens the linear solver tolerances and takes only a single Newton iteration. Calls [ARKStepSetDeltaGammaMax\(\)](#) to enforce Jacobian recomputation when the step size ratio changes by more than 100 times the unit roundoff (since nonlinear convergence is not tested). Only applicable when used in combination with the modified or inexact Newton iteration (not the fixed-point solver).

When  $f^I(t, y)$  is time-dependent, all linear solver structures (Jacobian, preconditioner) will be updated preceding *each* implicit stage. Thus one must balance the relative costs of such recomputation against the benefits of requiring only a single Newton linear solve.

Deprecated since version 6.1.0: Use [ARKodeSetLinear\(\)](#) instead.

int **ARKStepSetNonlinear**(void \*arkode\_mem)

Specifies that the implicit portion of the problem is nonlinear.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This is the default behavior of ARKStep, so the function is primarily useful to undo a previous call to [ARKStepSetLinear\(\)](#). Calls [ARKStepSetDeltaGammaMax\(\)](#) to reset the step size ratio threshold to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetNonlinear\(\)](#) instead.

int **ARKStepSetPredictorMethod**(void \*arkode\_mem, int method)

Specifies the method from §2.15.5 to use for predicting implicit solutions.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *method* – method choice ( $0 \leq \text{method} \leq 4$ ):
  - 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. **deprecated**
  - 5 is the minimum correction predictor, that uses all preceding stage information within the current step for prediction. **deprecated**

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default value is 0. If *method* is set to an undefined value, this default predictor will be used.

Options 4 and 5 are currently not supported when solving a problem involving a non-identity mass matrix. In that case, selection of *method* as 4 or 5 will instead default to the trivial predictor (*method* 0). **Both of these options have been deprecated, and will be removed from a future release.**

Deprecated since version 6.1.0: Use [ARKodeSetPredictorMethod\(\)](#) instead.

int **ARKStepSetStagePredictFn**(void \*arkode\_mem, [ARKStagePredictFn](#) PredictStage)

Sets the user-supplied function to update the implicit stage predictor prior to execution of the nonlinear or linear solver algorithms that compute the implicit stage solution.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

- *PredictStage* – name of user-supplied predictor function. If NULL, then any previously-provided stage prediction function will be disabled.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL

**Notes:**

See §5.4.6 for more information on this user-supplied routine.

Deprecated since version 6.1.0: Use [ARKodeSetStagePredictFn\(\)](#) instead.

int **ARKStepSetNlsRhsFn**(void \*arkode\_mem, [ARKRhsFn](#) nls\_fn)

Specifies an alternative implicit right-hand side function for evaluating  $f^I(t, y)$  within nonlinear system function evaluations (2.39) - (2.41).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nls\_fn* – the alternative C function for computing the right-hand side function  $f^I(t, y)$  in the ODE.

**Return value:**

- *ARK\_SUCCESS* if successful.
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL.

**Notes:**

The default is to use the implicit right-hand side function provided to [ARKStepCreate\(\)](#) in nonlinear system functions. If the input implicit right-hand side function is NULL, the default is used.

When using a non-default nonlinear solver, this function must be called *after* [ARKStepSetNonlinear-Solver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetNlsRhsFn\(\)](#) instead.

int **ARKStepSetMaxNonlinIters**(void \*arkode\_mem, int maxcor)

Specifies the maximum number of nonlinear solver iterations permitted per implicit stage solve within each time step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *maxcor* – maximum allowed solver iterations per stage ( $> 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value or if the SUNNONLINSOL module is NULL
- *ARK\_NLS\_OP\_ERR* if the SUNNONLINSOL object returned a failure flag

**Notes:**

The default value is 3; set *maxcor*  $\leq 0$  to specify this default.

Deprecated since version 6.1.0: Use [ARKodeSetMaxNonlinIters\(\)](#) instead.



int **ARKStepSetNonlinConvCoef**(void \*arkode\_mem, *sunrealtype* nlscoef)

Specifies the safety factor  $\epsilon$  used within the nonlinear solver convergence test (2.54).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nlscoef* – coefficient in nonlinear solver convergence test ( $> 0.0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default value is 0.1; set *nlscoef*  $\leq 0$  to specify this default.

Deprecated since version 6.1.0: Use [ARKodeSetNonlinConvCoef\(\)](#) instead.

int **ARKStepSetNonlinCRDown**(void \*arkode\_mem, *sunrealtype* crdown)

Specifies the constant  $c_r$  used in estimating the nonlinear solver convergence rate (2.53).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *crdown* – nonlinear convergence rate estimation constant (default is 0.3).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetNonlinCRDown\(\)](#) instead.

int **ARKStepSetNonlinRDiv**(void \*arkode\_mem, *sunrealtype* rdiv)

Specifies the nonlinear correction threshold  $r_{div}$  from (2.55), beyond which the iteration will be declared divergent.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *rdiv* – tolerance on nonlinear correction size ratio to declare divergence (default is 2.3).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetNonlinRDiv\(\)](#) instead.

int **ARKStepSetMaxConvFails**(void \*arkode\_mem, int maxncf)

Specifies the maximum number of nonlinear solver convergence failures permitted during one step,  $max_{ncf}$  from §2.15.3.1, before ARKStep will return with an error.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *maxncf* – maximum allowed nonlinear solver convergence failures per step ( $> 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default value is 10; set  $max_{ncf} \leq 0$  to specify this default.

Upon each convergence failure, ARKStep will first call the Jacobian setup routine and try again (if a Newton method is used). If a convergence failure still occurs, the time step size is reduced by the factor *etacf* (set within *ARKStepSetMaxCFailGrowth()*).

Deprecated since version 6.1.0: Use *ARKodeSetMaxConvFails()* instead.

int **ARKStepSetDeduceImplicitRhs**(void \*arkode\_mem, *sunbooleantype* deduce)

Specifies if implicit stage derivatives are deduced without evaluating  $f^I$ . See §2.15.1 for more details.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *deduce* – If *SUNFALSE* (default), the stage derivative is obtained by evaluating  $f^I$  with the stage solution returned from the nonlinear solver. If *SUNTRUE*, the stage derivative is deduced without an additional evaluation of  $f^I$ .

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL

Added in version 5.2.0.

Deprecated since version 6.1.0: Use *ARKodeSetDeduceImplicitRhs()* instead.

## Linear solver interface optional input functions

### Optional inputs for the ARKLS linear solver interface

int **ARKStepSetDeltaGammaMax**(void \*arkode\_mem, *sunrealtype* dgmax)

Specifies a scaled step size ratio tolerance,  $\Delta\gamma_{max}$  from §2.15.2.3, beyond which the linear solver setup routine will be signaled.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *dgmax* – tolerance on step size ratio change before calling linear solver setup routine (default is 0.2).

**Return value:**

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

**Notes:**

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use `ARKodeSetDeltaGammaMax()` instead.

int **ARKStepSetLSetupFrequency**(void \*arkode\_mem, int msbp)

Specifies the frequency of calls to the linear solver setup routine, *msbp* from §2.15.2.3.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *msbp* – the linear solver setup frequency.

**Return value:**

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

**Notes:**

Positive values of **msbp** specify the linear solver setup frequency. For example, an input of 1 means the setup function will be called every time step while an input of 2 means it will be called every other time step. If **msbp** is 0, the default value of 20 will be used. A negative value forces a linear solver step at each implicit stage.

Deprecated since version 6.1.0: Use `ARKodeSetLSetupFrequency()` instead.

int **ARKStepSetJacEvalFrequency**(void \*arkode\_mem, long int msbj)

Specifies the number of steps after which the Jacobian information is considered out-of-date, *msbj* from §2.15.2.3.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *msbj* – the Jacobian re-computation or preconditioner update frequency.

**Return value:**

- `ARKLS_SUCCESS` if successful.
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL.
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL.

**Notes:**

If *nstlj* is the step number at which the Jacobian information was last updated and *nst* is the current step number, *nst* - *nstlj* >= *msbj* indicates that the Jacobian information will be updated during the next linear solver setup call.

As the Jacobian update frequency is only checked *within* calls to the linear solver setup routine, Jacobian information may be more than *msbj* steps old when updated depending on when a linear solver setup call occurs. See §2.15.2.3 for more information on when linear solver setups are performed.

Passing a value *msbj* ≤ 0 indicates to use the default value of 51.

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

Deprecated since version 6.1.0: Use [ARKodeSetJacEvalFrequency\(\)](#) instead.

### Optional inputs for matrix-based SUNLinearSolver modules

int **ARKStepSetJacFn**(void \*arkode\_mem, [ARKLsJacFn](#) jac)

Specifies the Jacobian approximation routine to be used for the matrix-based solver with the ARKLS interface.

#### Arguments:

- *arkode\_mem* – pointer to the ARKStep memory block.
- *jac* – name of user-supplied Jacobian approximation function.

#### Return value:

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

#### Notes:

This routine must be called after the ARKLS linear solver interface has been initialized through a call to [ARKStepSetLinearSolver\(\)](#).

By default, ARKLS uses an internal difference quotient function for the [SUNMATRIX\\_DENSE](#) and [SUNMATRIX\\_BAND](#) modules. If NULL is passed in for *jac*, this default is used. An error will occur if no *jac* is supplied when using other matrix types.

The function type [ARKLsJacFn\(\)](#) is described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetJacFn\(\)](#) instead.

int **ARKStepSetLinSysFn**(void \*arkode\_mem, [ARKLsLinSysFn](#) linsys)

Specifies the linear system approximation routine to be used for the matrix-based solver with the ARKLS interface.

#### Arguments:

- *arkode\_mem* – pointer to the ARKStep memory block.
- *linsys* – name of user-supplied linear system approximation function.

#### Return value:

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

#### Notes:

This routine must be called after the ARKLS linear solver interface has been initialized through a call to [ARKStepSetLinearSolver\(\)](#).

By default, ARKLS uses an internal linear system function that leverages the SUNMATRIX API to form the system  $M - \gamma J$ . If NULL is passed in for *linsys*, this default is used.

The function type [ARKLsLinSysFn\(\)](#) is described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetLinSysFn\(\)](#) instead.

int **ARKStepSetMassFn**(void \*arkode\_mem, *ARKLsMassFn* mass)

Specifies the mass matrix approximation routine to be used for the matrix-based solver with the ARKLS interface.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *mass* – name of user-supplied mass matrix approximation function.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_MASSMEM\_NULL* if the mass matrix solver memory was NULL
- *ARKLS\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This routine must be called after the ARKLS mass matrix solver interface has been initialized through a call to *ARKStepSetMassLinearSolver()*.

Since there is no default difference quotient function for mass matrices, *mass* must be non-NULL.

The function type *ARKLsMassFn()* is described in §5.4.

Deprecated since version 6.1.0: Use *ARKodeSetMassFn()* instead.

int **ARKStepSetLinearSolutionScaling**(void \*arkode\_mem, *sunbooleantype* onoff)

Enables or disables scaling the linear system solution to account for a change in  $\gamma$  in the linear system. For more details see §10.2.1.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *onoff* – flag to enable (SUNTRUE) or disable (SUNFALSE) scaling

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_ILL\_INPUT* if the attached linear solver is not matrix-based

**Notes:**

Linear solution scaling is enabled by default when a matrix-based linear solver is attached.

Deprecated since version 6.1.0: Use *ARKodeSetLinearSolutionScaling()* instead.

### Optional inputs for matrix-free SUNLinearSolver modules

int **ARKStepSetJacTimes**(void \*arkode\_mem, *ARKLsJacTimesSetupFn* jtsetup, *ARKLsJacTimesVecFn* jtimes)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *jtsetup* – user-defined Jacobian-vector setup function. Pass NULL if no setup is necessary.
- *jtimes* – user-defined Jacobian-vector product function.

**Return value:**

- `ARKLS_SUCCESS` if successful.
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL.
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL.
- `ARKLS_ILL_INPUT` if an input has an illegal value.
- `ARKLS_SUNLS_FAIL` if an error occurred when setting up the Jacobian-vector product in the SUN-LinearSolver object used by the ARKLS interface.

**Notes:**

The default is to use an internal finite difference quotient for *jtimes* and to leave out *jtsetup*. If NULL is passed to *jtimes*, these defaults are used. A user may specify non-NULL *jtimes* and NULL *jtsetup* inputs.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [ARKStepSetLinearSolver\(\)](#).

The function types [ARKLSJacTimesSetupFn](#) and [ARKLSJacTimesVecFn](#) are described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetJacTimes\(\)](#) instead.

int **ARKStepSetJacTimesRhsFn**(void \*arkode\_mem, [ARKRhsFn](#) jtimesRhsFn)

Specifies an alternative implicit right-hand side function for use in the internal Jacobian-vector product difference quotient approximation.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *jtimesRhsFn* – the name of the C function (of type [ARKRhsFn\(\)](#)) defining the alternative right-hand side function.

**Return value:**

- `ARKLS_SUCCESS` if successful.
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL.
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL.
- `ARKLS_ILL_INPUT` if an input has an illegal value.

**Notes:**

The default is to use the implicit right-hand side function provided to [ARKStepCreate\(\)](#) in the internal difference quotient. If the input implicit right-hand side function is NULL, the default is used.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [ARKStepSetLinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetJacTimesRhsFn\(\)](#) instead.

int **ARKStepSetMassTimes**(void \*arkode\_mem, [ARKLsMassTimesSetupFn](#) mtsetup, [ARKLsMassTimesVecFn](#) mtimes, void \*mtimes\_data)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *mtsetup* – user-defined mass matrix-vector setup function. Pass NULL if no setup is necessary.
- *mtimes* – user-defined mass matrix-vector product function.
- *mtimes\_data* – a pointer to user data, that will be supplied to both the *mtsetup* and *mtimes* functions.

**Return value:**

- `ARKLS_SUCCESS` if successful.
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL.
- `ARKLS_MASSMEM_NULL` if the mass matrix solver memory was NULL.
- `ARKLS_ILL_INPUT` if an input has an illegal value.
- `ARKLS_SUNLS_FAIL` if an error occurred when setting up the mass-matrix-vector product in the `SUNLinearSolver` object used by the ARKLS interface.

**Notes:**

There is no default finite difference quotient for *mtimes*, so if using the ARKLS mass matrix solver interface with NULL-valued `SUNMATRIX` input *M*, and this routine is called with NULL-valued *mtimes*, an error will occur. A user may specify NULL for *mtsetup*.

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

The function types `ARKLSMassTimesSetupFn` and `ARKLSMassTimesVecFn` are described in §5.4.

Deprecated since version 6.1.0: Use `ARKodeSetMassTimes()` instead.

**Optional inputs for iterative `SUNLinearSolver` modules**

int **ARKStepSetPreconditioner**(void \*arkode\_mem, `ARKLsPrecSetupFn` psetup, `ARKLsPrecSolveFn` psolve)

Specifies the user-supplied preconditioner setup and solve functions.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *psetup* – user defined preconditioner setup function. Pass NULL if no setup is needed.
- *psolve* – user-defined preconditioner solve function.

**Return value:**

- `ARKLS_SUCCESS` if successful.
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL.
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL.
- `ARKLS_ILL_INPUT` if an input has an illegal value.
- `ARKLS_SUNLS_FAIL` if an error occurred when setting up preconditioning in the `SUNLinearSolver` object used by the ARKLS interface.

**Notes:**

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

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

Both of the function types `ARKLsPrecSetupFn()` and `ARKLsPrecSolveFn()` are described in §5.4.

Deprecated since version 6.1.0: Use `ARKodeSetPreconditioner()` instead.

int **ARKStepSetMassPreconditioner**(void \*arkode\_mem, `ARKLsMassPrecSetupFn` psetup, `ARKLsMassPrecSolveFn` psolve)

Specifies the mass matrix preconditioner setup and solve functions.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *psetup* – user defined preconditioner setup function. Pass NULL if no setup is to be done.
- *psolve* – user-defined preconditioner solve function.

**Return value:**

- *ARKLS\_SUCCESS* if successful.
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL.
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL.
- *ARKLS\_ILL\_INPUT* if an input has an illegal value.
- *ARKLS\_SUNLS\_FAIL* if an error occurred when setting up preconditioning in the SUNLinearSolver object used by the ARKLS interface.

**Notes:**

This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to [ARKStepSetMassLinearSolver\(\)](#).

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

Both of the function types [ARKLSMassPrecSetupFn\(\)](#) and [ARKLSMassPrecSolveFn\(\)](#) are described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetMassPreconditioner\(\)](#) instead.

int **ARKStepSetEpsLin**(void \*arkode\_mem, *sunrealtype* eplifac)

Specifies the factor  $\epsilon_L$  by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the linear iteration.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *eplifac* – linear convergence safety factor.

**Return value:**

- *ARKLS\_SUCCESS* if successful.
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL.
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL.
- *ARKLS\_ILL\_INPUT* if an input has an illegal value.

**Notes:**

Passing a value  $eplifac \leq 0$  indicates to use the default value of 0.05.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [ARKStepSetLinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetEpsLin\(\)](#) instead.

int **ARKStepSetMassEpsLin**(void \*arkode\_mem, *sunrealtype* eplifac)

Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the mass matrix linear iteration.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *eplifac* – linear convergence safety factor.



**Return value:**

- `ARKLS_SUCCESS` if successful.
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL.
- `ARKLS_MASSMEM_NULL` if the mass matrix solver memory was NULL.
- `ARKLS_ILL_INPUT` if an input has an illegal value.

**Notes:**

This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to [`ARKStepSetMassLinearSolver\(\)`](#).

Passing a value  $eplifac \leq 0$  indicates to use the default value of 0.05.

Deprecated since version 6.1.0: Use [`ARKodeSetMassEpsLin\(\)`](#) instead.

int **ARKStepSetLSNormFactor**(void \*arkode\_mem, *sunrealtype* nrmfac)

Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for Newton linear system solves.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nrmfac* – the norm conversion factor. If *nrmfac* is:
  - > 0 then the provided value is used.
  - = 0 then the conversion factor is computed using the vector length i.e.,  $nrmfac = \sqrt{N - VGetLength(y)}$  (default).
  - < 0 then the conversion factor is computed using the vector dot product i.e.,  $nrmfac = \sqrt{N - VDotProd(v, v)}$  where all the entries of *v* are one.

**Return value:**

- `ARK_SUCCESS` if successful.
- `ARK_MEM_NULL` if the ARKStep memory was NULL.

**Notes:**

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [`ARKStepSetLinearSolver\(\)`](#).

Deprecated since version 6.1.0: Use [`ARKodeSetLSNormFactor\(\)`](#) instead.

int **ARKStepSetMassLSNormFactor**(void \*arkode\_mem, *sunrealtype* nrmfac)

Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for mass matrix linear system solves.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nrmfac* – the norm conversion factor. If *nrmfac* is:
  - > 0 then the provided value is used.
  - = 0 then the conversion factor is computed using the vector length i.e.,  $nrmfac = \sqrt{N - VGetLength(y)}$  (default).
  - < 0 then the conversion factor is computed using the vector dot product i.e.,  $nrmfac = \sqrt{N - VDotProd(v, v)}$  where all the entries of *v* are one.

**Return value:**

- `ARK_SUCCESS` if successful.
- `ARK_MEM_NULL` if the ARKStep memory was NULL.

**Notes:**

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

Deprecated since version 6.1.0: Use `ARKodeSetMassLSNormFactor()` instead.

**Rootfinding optional input functions**

int **ARKStepSetRootDirection**(void \*arkode\_mem, int \*rootdir)

Specifies the direction of zero-crossings to be located and returned.

**Arguments:**

- `arkode_mem` – pointer to the ARKStep memory block.
- `rootdir` – state array of length `nrtfn`, the number of root functions  $g_i$  (the value of `nrtfn` was supplied in the call to `ARKStepRootInit()`). If `rootdir[i] == 0` then crossing in either direction for  $g_i$  should be reported. A value of +1 or -1 indicates that the solver should report only zero-crossings where  $g_i$  is increasing or decreasing, respectively.

**Return value:**

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

**Notes:**

The default behavior is to monitor for both zero-crossing directions.

Deprecated since version 6.1.0: Use `ARKodeSetRootDirection()` instead.

int **ARKStepSetNoInactiveRootWarn**(void \*arkode\_mem)

Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

**Arguments:**

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

**Return value:**

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

**Notes:**

ARKStep will not report the initial conditions as a possible zero-crossing (assuming that one or more components  $g_i$  are zero at the initial time). However, if it appears that some  $g_i$  is identically zero at the initial time (i.e.,  $g_i$  is zero at the initial time *and* after the first step), ARKStep will issue a warning which can be disabled with this optional input function.

Deprecated since version 6.1.0: Use `ARKodeSetNoInactiveRootWarn()` instead.

### 5.7.1.9 Interpolated output function

int **ARKStepGetDky**(void \*arkode\_mem, *sunrealtype* t, int k, *N\_Vector* dky)

Computes the  $k$ -th derivative of the function  $y$  at the time  $t$ , i.e.  $y^{(k)}(t)$ , 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. This routine uses an interpolating polynomial of degree  $\min(\text{degree}, 5)$ , where *degree* is the argument provided to [ARKStepSetInterpolantDegree\(\)](#). The user may request  $k$  in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$  where  $kmax$  depends on the choice of interpolation module. For Hermite interpolants  $kmax = 5$  and for Lagrange interpolants  $kmax = 3$ .

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- $t$  – the value of the independent variable at which the derivative is to be evaluated.
- $k$  – the derivative order requested.
- *dky* – output vector (must be allocated by the user).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_BAD\_K* if  $k$  is not in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$ .
- *ARK\_BAD\_T* if  $t$  is not in the interval  $[t_n - h_n, t_n]$
- *ARK\_BAD\_DKY* if the *dky* vector was NULL
- *ARK\_MEM\_NULL* if the ARKStep memory is NULL

**Notes:**

It is only legal to call this function after a successful return from [ARKStepEvolve\(\)](#).

A user may access the values  $t_n$  and  $h_n$  via the functions [ARKStepGetCurrentTime\(\)](#) and [ARKStepGetLastStep\(\)](#), respectively.

Deprecated since version 6.1.0: Use [ARKodeGetDky\(\)](#) instead.

### 5.7.1.10 Optional output functions

#### Main solver optional output functions

int **ARKStepGetWorkspace**(void \*arkode\_mem, long int \*lenrw, long int \*leniw)

Returns the ARKStep real and integer workspace sizes.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *lenrw* – the number of *sunrealtype* values in the ARKStep workspace.
- *leniw* – the number of integer values in the ARKStep workspace.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetWorkspace\(\)](#) instead.

int **ARKStepGetNumSteps**(void \*arkode\_mem, long int \*nsteps)

Returns the cumulative number of internal steps taken by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nsteps* – number of steps taken in the solver.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumSteps\(\)](#) instead.

int **ARKStepGetActualInitStep**(void \*arkode\_mem, *sunrealtype* \*hinused)

Returns the value of the integration step size used on the first step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *hinused* – actual value of initial step size.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

Even if the value of the initial integration step was specified by the user through a call to [ARKStepSetInitStep\(\)](#), this value may have been changed by ARKStep to ensure that the step size fell within the prescribed bounds ( $h_{min} \leq h_0 \leq h_{max}$ ), or to satisfy the local error test condition, or to ensure convergence of the nonlinear solver.

Deprecated since version 6.1.0: Use [ARKodeGetActualInitStep\(\)](#) instead.

int **ARKStepGetLastStep**(void \*arkode\_mem, *sunrealtype* \*hlast)

Returns the integration step size taken on the last successful internal step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *hlast* – step size taken on the last internal step.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetLastStep\(\)](#) instead.

int **ARKStepGetCurrentStep**(void \*arkode\_mem, *sunrealtype* \*hcur)

Returns the integration step size to be attempted on the next internal step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *hcur* – step size to be attempted on the next internal step.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentStep\(\)](#) instead.

int **ARKStepGetCurrentTime**(void \*arkode\_mem, *sunrealtype* \*tcur)

Returns the current internal time reached by the solver.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *tcur* – current internal time reached.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentTime\(\)](#) instead.

int **ARKStepGetCurrentState**(void \*arkode\_mem, *N\_Vector* \*ycur)

Returns the current internal solution reached by the solver.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *ycur* – current internal solution.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

Users should exercise extreme caution when using this function, as altering values of *ycur* may lead to undesirable behavior, depending on the particular use case and on when this routine is called.

Deprecated since version 6.1.0: Use [ARKodeGetCurrentState\(\)](#) instead.

int **ARKStepGetCurrentGamma**(void \*arkode\_mem, *sunrealtype* \*gamma)

Returns the current internal value of  $\gamma$  used in the implicit solver Newton matrix (see equation (2.47)).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *gamma* – current step size scaling factor in the Newton system.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentGamma\(\)](#) instead.

int **ARKStepGetTolScaleFactor**(void \*arkode\_mem, *sunrealtype* \*tolsfac)

Returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.

- *tol<sub>sfac</sub>* – suggested scaling factor for user-supplied tolerances.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetTolScaleFactor\(\)](#) instead.

int **ARKStepGetErrWeights**(void \*arkode\_mem, *N\_Vector* eweight)

Returns the current error weight vector.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *eweight* – solution error weights at the current time.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

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

Deprecated since version 6.1.0: Use [ARKodeGetErrWeights\(\)](#) instead.

int **ARKStepGetResWeights**(void \*arkode\_mem, *N\_Vector* rweight)

Returns the current residual weight vector.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *rweight* – residual error weights at the current time.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

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

Deprecated since version 6.1.0: Use [ARKodeGetResWeights\(\)](#) instead.

int **ARKStepGetStepStats**(void \*arkode\_mem, long int \*nsteps, *sunrealtype* \*hinused, *sunrealtype* \*hlast, *sunrealtype* \*hcur, *sunrealtype* \*tcur)

Returns many of the most useful optional outputs in a single call.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nsteps* – number of steps taken in the solver.
- *hinused* – actual value of initial step size.
- *hlast* – step size taken on the last internal step.
- *hcur* – step size to be attempted on the next internal step.
- *tcur* – current internal time reached.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetStepStats\(\)](#) instead.

int **ARKStepPrintAllStats**(void \*arkode\_mem, FILE \*outfile, *SUNOutputFormat* fmt)

Outputs all of the integrator, nonlinear solver, linear solver, and other statistics.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *outfile* – pointer to output file.
- *fmt* – the output format:
  - *SUN\_OUTPUTFORMAT\_TABLE* – prints a table of values
  - *SUN\_OUTPUTFORMAT\_CSV* – prints a comma-separated list of key and value pairs e.g., key1, value1, key2, value2, ...

**Return value:**

- *ARK\_SUCCESS* – if the output was successfully.
- *ARK\_MEM\_NULL* – if the ARKStep memory was NULL.
- *ARK\_ILL\_INPUT* – if an invalid formatting option was provided.

**Note**

The Python module `tools/suntools` provides utilities to read and output the data from a SUNDIALS CSV output file using the key and value pair format.

Added in version 5.2.0.

Deprecated since version 6.1.0: Use [ARKodePrintAllStats\(\)](#) instead.

char \***ARKStepGetReturnFlagName**(long int flag)

Returns the name of the ARKStep constant corresponding to *flag*. See [ARKODE Constants](#).

**Arguments:**

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

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

**Warning**

The user is responsible for freeing the returned string.

Deprecated since version 6.1.0: Use [ARKodeGetReturnFlagName\(\)](#) instead.

int **ARKStepGetNumExpSteps**(void \*arkode\_mem, long int \*expsteps)

Returns the cumulative number of stability-limited steps taken by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumExpSteps\(\)](#) instead.

int **ARKStepGetNumAccSteps**(void \*arkode\_mem, long int \*accsteps)

Returns the cumulative number of accuracy-limited steps taken by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *accsteps* – number of accuracy-limited steps taken in the solver.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumAccSteps\(\)](#) instead.

int **ARKStepGetNumStepAttempts**(void \*arkode\_mem, long int \*step\_attempts)

Returns the cumulative number of steps attempted by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *step\_attempts* – number of steps attempted by solver.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumStepAttempts\(\)](#) instead.

int **ARKStepGetNumRhsEvals**(void \*arkode\_mem, long int \*nfe\_evals, long int \*nfi\_evals)

Returns the number of calls to the user's right-hand side functions,  $f^E$  and  $f^I$  (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nfe\_evals* – number of calls to the user's  $f^E(t, y)$  function.
- *nfi\_evals* – number of calls to the user's  $f^I(t, y)$  function.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

The *nfi\_evals* value does not account for calls made to  $f^I$  by a linear solver or preconditioner module.

Deprecated since version 6.2.0: Use [ARKodeGetNumRhsEvals\(\)](#) instead.

int **ARKStepGetNumErrTestFails**(void \*arkode\_mem, long int \*netfails)

Returns the number of local error test failures that have occurred (so far).

**Arguments:**



- *arkode\_mem* – pointer to the ARKStep memory block.
- *netfails* – number of error test failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumErrTestFails\(\)](#) instead.

int **ARKStepGetNumStepSolveFails**(void \*arkode\_mem, long int \*ncnf)

Returns the number of failed steps due to a nonlinear solver failure (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *ncnf* – number of step failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumStepSolveFails\(\)](#) instead.

int **ARKStepGetCurrentButcherTables**(void \*arkode\_mem, *ARKodeButcherTable* \*Bi, *ARKodeButcherTable* \*Be)

Returns the explicit and implicit Butcher tables currently in use by the solver.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *Bi* – pointer to the implicit Butcher table structure.
- *Be* – pointer to the explicit Butcher table structure.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Note:** The [ARKodeButcherTable](#) data structure is defined as a pointer to the following C structure:

```
typedef struct ARKStepButcherTableMem {
    int q;           /* method order of accuracy */
    int p;           /* embedding order of accuracy */
    int stages;      /* number of stages */
    sunrealtype **A; /* Butcher table coefficients */
    sunrealtype *c;  /* canopy node coefficients */
    sunrealtype *b;  /* root node coefficients */
    sunrealtype *d;  /* embedding coefficients */
} *ARKStepButcherTable;
```

For more details see §6.

int **ARKStepGetEstLocalErrors**(void \*arkode\_mem, *N\_Vector* ele)

Returns the vector of estimated local truncation errors for the current step.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *ele* – vector of estimated local truncation errors.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

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

The values returned in *ele* are valid only after a successful call to [ARKStepEvolve\(\)](#) (i.e., it returned a non-negative value).

The *ele* vector, together with the *eweight* vector from [ARKStepGetErrWeights\(\)](#), can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the WRMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as *eweight[i]\*ele[i]*.

Deprecated since version 6.1.0: Use [ARKodeGetEstLocalErrors\(\)](#) instead.

int **ARKStepGetTimestepperStats**(void \*arkode\_mem, long int \*expsteps, long int \*accsteps, long int \*step\_attempts, long int \*nfe\_evals, long int \*nfi\_evals, long int \*nlinsetups, long int \*netfails)

Returns many of the most useful time-stepper statistics in a single call.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.
- *accsteps* – number of accuracy-limited steps taken in the solver.
- *step\_attempts* – number of steps attempted by the solver.
- *nfe\_evals* – number of calls to the user's  $f^E(t, y)$  function.
- *nfi\_evals* – number of calls to the user's  $f^I(t, y)$  function.
- *nlinsetups* – number of linear solver setup calls made.
- *netfails* – number of error test failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

int **ARKStepGetNumConstrFails**(void \*arkode\_mem, long int \*nconstrfails)

Returns the cumulative number of constraint test failures (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nconstrfails* – number of constraint test failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumConstrFails\(\)](#) instead.

int **ARKStepGetUserData**(void \*arkode\_mem, void \*\*user\_data)

Returns the user data pointer previously set with [ARKStepSetUserData\(\)](#).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *user\_data* – memory reference to a user data pointer

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

Added in version 5.3.0.

Deprecated since version 6.1.0: Use [ARKodeGetUserData\(\)](#) instead.

**Implicit solver optional output functions**

int **ARKStepGetNumLinSolvSetups**(void \*arkode\_mem, long int \*nlinsetups)

Returns the number of calls made to the linear solver's setup routine (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nlinsetups* – number of linear solver setup calls made.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Note:** This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumLinSolvSetups\(\)](#) instead.

int **ARKStepGetNumNonlinSolvIters**(void \*arkode\_mem, long int \*nniters)

Returns the number of nonlinear solver iterations performed (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nniters* – number of nonlinear iterations performed.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NLS\_OP\_ERR* if the SUNNONLINSOL object returned a failure flag

**Note:** This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumNonlinSolvIters\(\)](#) instead.

int **ARKStepGetNumNonlinSolvConvFails**(void \*arkode\_mem, long int \*nncfails)

Returns the number of nonlinear solver convergence failures that have occurred (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nncfails* – number of nonlinear convergence failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Note:** This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumNonlinSolvConvFails\(\)](#) instead.

int **ARKStepGetNonlinSolvStats**(void \*arkode\_mem, long int \*nniters, long int \*nncfails)

Returns all of the nonlinear solver statistics in a single call.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nniters* – number of nonlinear iterations performed.
- *nncfails* – number of nonlinear convergence failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NLS\_OP\_ERR* if the SUNNONLINSOL object returned a failure flag

**Note:** This is only accumulated for the “life” of the nonlinear solver object; the counters are reset whenever a new nonlinear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNonlinSolvStats\(\)](#) instead.

## Rootfinding optional output functions

int **ARKStepGetRootInfo**(void \*arkode\_mem, int \*rootsfound)

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

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *rootsfound* – array of length *nrtfn* with the indices of the user functions  $g_i$  found to have a root (the value of *nrtfn* was supplied in the call to [ARKStepRootInit\(\)](#)). For  $i = 0 \dots nrtfn-1$ , *rootsfound*[*i*] is nonzero if  $g_i$  has a root, and 0 if not.

**Return value:**

- *ARK\_SUCCESS* if successful

- `ARK_MEM_NULL` if the ARKStep memory was NULL

**Notes:**

The user must allocate space for *rootsfound* prior to calling this function.

For the components of  $g_i$  for which a root was found, the sign of `rootsfound[i]` indicates the direction of zero-crossing. A value of +1 indicates that  $g_i$  is increasing, while a value of -1 indicates a decreasing  $g_i$ .

Deprecated since version 6.1.0: Use [ARKodeGetRootInfo\(\)](#) instead.

int **ARKStepGetNumGEvals**(void \*arkode\_mem, long int \*ngevals)

Returns the cumulative number of calls made to the user's root function  $g$ .

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *ngevals* – number of calls made to  $g$  so far.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ARKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumGEvals\(\)](#) instead.

**Linear solver interface optional output functions**

int **ARKStepGetJac**(void \*arkode\_mem, [SUNMatrix](#) \*J)

Returns the internally stored copy of the Jacobian matrix of the ODE implicit right-hand side function.

**Parameters**

- *arkode\_mem* – the ARKStep memory structure
- *J* – the Jacobian matrix

**Return values**

- `ARKLS_SUCCESS` – the output value has been successfully set
- `ARKLS_MEM_NULL` – *arkode\_mem* was NULL
- `ARKLS_LMEM_NULL` – the linear solver interface has not been initialized

**Warning**

This function is provided for debugging purposes and the values in the returned matrix should not be altered.

Deprecated since version 6.1.0: Use [ARKodeGetJac\(\)](#) instead.

int **ARKStepGetJacTime**(void \*arkode\_mem, [sunrealtype](#) \*t\_J)

Returns the time at which the internally stored copy of the Jacobian matrix of the ODE implicit right-hand side function was evaluated.

**Parameters**

- *arkode\_mem* – the ARKStep memory structure
- *t\_J* – the time at which the Jacobian was evaluated

**Return values**

- **ARKLS\_SUCCESS** – the output value has been successfully set
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized

Deprecated since version 6.1.0: Use [`ARKodeGetJacTime\(\)`](#) instead.

int **ARKStepGetJacNumSteps**(void \*arkode\_mem, long int \*nst\_J)

Returns the value of the internal step counter at which the internally stored copy of the Jacobian matrix of the ODE implicit right-hand side function was evaluated.

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **nst\_J** – the value of the internal step counter at which the Jacobian was evaluated

**Return values**

- **ARKLS\_SUCCESS** – the output value has been successfully set
- **ARKLS\_MEM\_NULL** – `arkode_mem` was NULL
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized

Deprecated since version 6.1.0: Use [`ARKodeGetJacNumSteps\(\)`](#) instead.

int **ARKStepGetLinWorkSpace**(void \*arkode\_mem, long int \*lenrwLS, long int \*leniwLS)

Returns the real and integer workspace used by the ARKLS linear solver interface.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *lenrwLS* – the number of `sunrealtype` values in the ARKLS workspace.
- *leniwLS* – the number of integer values in the ARKLS workspace.

**Return value:**

- **ARKLS\_SUCCESS** if successful
- **ARKLS\_MEM\_NULL** if the ARKStep memory was NULL
- **ARKLS\_LMEM\_NULL** if the linear solver memory was NULL

**Notes:**

The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the `SUNLinearSolver` object attached to it. The template Jacobian matrix allocated by the user outside of ARKLS is not included in this report.

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

Deprecated since version 6.1.0: Use [`ARKodeGetLinWorkSpace\(\)`](#) instead.

int **ARKStepGetNumJacEvals**(void \*arkode\_mem, long int \*njevals)

Returns the number of Jacobian evaluations.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *njevals* – number of Jacobian evaluations.

**Return value:**

- **ARKLS\_SUCCESS** if successful

- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumJacEvals\(\)`](#) instead.

int **ARKStepGetNumPrecEvals**(void \*arkode\_mem, long int \*npevals)

Returns the total number of preconditioner evaluations, i.e. the number of calls made to *psetup* with *jok* = *SUNFALSE* and that returned *\*jcurPtr* = *SUNTRUE*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *npevals* – the current number of calls to *psetup*.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumPrecEvals\(\)`](#) instead.

int **ARKStepGetNumPrecSolves**(void \*arkode\_mem, long int \*npsolves)

Returns the number of calls made to the preconditioner solve function, *psolve*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *npsolves* – the number of calls to *psolve*.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumPrecSolves\(\)`](#) instead.

int **ARKStepGetNumLinIters**(void \*arkode\_mem, long int \*nliters)

Returns the cumulative number of linear iterations.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nliters* – the current number of linear iterations.

**Return value:**

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

- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumLinIters\(\)`](#) instead.

int **ARKStepGetNumLinConvFails**(void \*arkode\_mem, long int \*nlcfails)

Returns the cumulative number of linear convergence failures.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nlcfails* – the current number of linear convergence failures.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumLinConvFails\(\)`](#) instead.

int **ARKStepGetNumJTSetupEvals**(void \*arkode\_mem, long int \*njtsetup)

Returns the cumulative number of calls made to the user-supplied Jacobian-vector setup function, *jtsetup*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *jtsetup* – the current number of calls to *jtsetup*.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumJTSetupEvals\(\)`](#) instead.

int **ARKStepGetNumJtimesEvals**(void \*arkode\_mem, long int \*njvevals)

Returns the cumulative number of calls made to the Jacobian-vector product function, *jtimes*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *njvevals* – the current number of calls to *jtimes*.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL



**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumJtimesEvals\(\)](#) instead.

int **ARKStepGetNumLinRhsevals**(void \*arkode\_mem, long int \*nfevalsLS)

Returns the number of calls to the user-supplied implicit right-hand side function  $f^I$  for finite difference Jacobian or Jacobian-vector product approximation.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nfevalsLS* – the number of calls to the user implicit right-hand side function.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Notes:**

The value *nfevalsLS* is incremented only if the default internal difference quotient function is used.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumLinRhsevals\(\)](#) instead.

int **ARKStepGetLastLinFlag**(void \*arkode\_mem, long int \*lsflag)

Returns the last return value from an ARKLS routine.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *lsflag* – the value of the last return flag from an ARKLS function.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Notes:**

If the ARKLS setup function failed when using the SUNLINSOL\_DENSE or SUNLINSOL\_BAND modules, then the value of *lsflag* is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix. For all other failures, *lsflag* is negative.

Otherwise, if the ARKLS setup function failed ([ARKStepEvolve\(\)](#) returned *ARK\_LSETUP\_FAIL*), then *lsflag* will be *SUNLS\_PSET\_FAIL\_UNREC*, *SUNLS\_ASET\_FAIL\_UNREC* or *SUN\_ERR\_EXT\_FAIL*.

If the ARKLS solve function failed ([ARKStepEvolve\(\)](#) returned *ARK\_LSOLVE\_FAIL*), then *lsflag* contains the error return flag from the SUNLinearSolver object, which will be one of: *SUN\_ERR\_ARG\_CORRUPT*, indicating that the SUNLinearSolver memory is NULL; *SUNLS\_ATIMES\_NULL*, indicating that a matrix-free iterative solver was provided, but is missing a routine for the matrix-vector product approximation, *SUNLS\_ATIMES\_FAIL\_UNREC*, indicating an unrecoverable failure in the  $Jv$  function; *SUNLS\_PSOLVE\_NULL*, indicating that an iterative linear solver was configured to use preconditioning, but no preconditioner solve routine was provided, *SUNLS\_PSOLVE\_FAIL\_UNREC*, indicating that the

preconditioner solve function failed unrecoverably; *SUNLS\_GS\_FAIL*, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only); *SUNLS\_QRSOL\_FAIL*, indicating that the matrix *R* was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or *SUN\_ERR\_EXT\_FAIL*, indicating an unrecoverable failure in an external iterative linear solver package.

Deprecated since version 6.1.0: Use [ARKodeGetLastLinFlag\(\)](#) instead.

char \***ARKStepGetLinReturnFlagName**(long int lsflag)

Returns the name of the ARKLS constant corresponding to *lsflag*.

**Arguments:**

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

**Return value:** The return value is a string containing the name of the corresponding constant. If using the SUNLINSOL\_DENSE or SUNLINSOL\_BAND modules, then if  $1 \leq lsflag \leq n$  (LU factorization failed), this routine returns “NONE”.

**Warning**

The user is responsible for freeing the returned string.

Deprecated since version 6.1.0: Use [ARKodeGetLinReturnFlagName\(\)](#) instead.

int **ARKStepGetMassWorkspace**(void \*arkode\_mem, long int \*lenrwMLS, long int \*leniwMLS)

Returns the real and integer workspace used by the ARKLS mass matrix linear solver interface.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *lenrwMLS* – the number of `sunrealtype` values in the ARKLS mass solver workspace.
- *leniwMLS* – the number of integer values in the ARKLS mass solver workspace.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Notes:**

The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the `SUNLinearSolver` object attached to it. The template mass matrix allocated by the user outside of ARKLS is not included in this report.

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

Deprecated since version 6.1.0: Use [ARKodeGetMassWorkspace\(\)](#) instead.

int **ARKStepGetNumMassSetups**(void \*arkode\_mem, long int \*nmsetups)

Returns the number of calls made to the ARKLS mass matrix solver ‘setup’ routine; these include all calls to the user-supplied mass-matrix constructor function.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmsetups* – number of calls to the mass matrix solver setup routine.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [\*ARKodeGetNumMassSetups\(\)\*](#) instead.

int **ARKStepGetNumMassMultSetups**(void \*arkode\_mem, long int \*nmvsetups)

Returns the number of calls made to the ARKLS mass matrix ‘matvec setup’ (matrix-based solvers) routine.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmvsetups* – number of calls to the mass matrix matrix-times-vector setup routine.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [\*ARKodeGetNumMassMultSetups\(\)\*](#) instead.

int **ARKStepGetNumMassMult**(void \*arkode\_mem, long int \*nmmults)

Returns the number of calls made to the ARKLS mass matrix ‘matvec’ routine (matrix-based solvers) or the user-supplied *mtimes* routine (matrix-free solvers).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmmults* – number of calls to the mass matrix solver matrix-times-vector routine.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [\*ARKodeGetNumMassMult\(\)\*](#) instead.

int **ARKStepGetNumMassSolves**(void \*arkode\_mem, long int \*nmsolves)

Returns the number of calls made to the ARKLS mass matrix solver ‘solve’ routine.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmsolves* – number of calls to the mass matrix solver solve routine.

**Return value:**

- *ARKLS\_SUCCESS* if successful

- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumMassSolves\(\)`](#) instead.

int **ARKStepGetNumMassPrecEvals**(void \*arkode\_mem, long int \*nmpevals)

Returns the total number of mass matrix preconditioner evaluations, i.e. the number of calls made to *psetup*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmpevals* – the current number of calls to *psetup*.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumMassPrecEvals\(\)`](#) instead.

int **ARKStepGetNumMassPrecSolves**(void \*arkode\_mem, long int \*nmpsolves)

Returns the number of calls made to the mass matrix preconditioner solve function, *psolve*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmpsolves* – the number of calls to *psolve*.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumMassPrecSolves\(\)`](#) instead.

int **ARKStepGetNumMassIters**(void \*arkode\_mem, long int \*nmiters)

Returns the cumulative number of mass matrix solver iterations.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmiters* – the current number of mass matrix solver linear iterations.

**Return value:**

- `ARKLS_SUCCESS` if successful
- `ARKLS_MEM_NULL` if the ARKStep memory was NULL
- `ARKLS_LMEM_NULL` if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumMassIters\(\)](#) instead.

int **ARKStepGetNumMassConvFails**(void \*arkode\_mem, long int \*nmcfails)

Returns the cumulative number of mass matrix solver convergence failures.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmcfails* – the current number of mass matrix solver convergence failures.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumMassConvFails\(\)](#) instead.

int **ARKStepGetNumMTSetups**(void \*arkode\_mem, long int \*nmtsetup)

Returns the cumulative number of calls made to the user-supplied mass-matrix-vector product setup function, *mtsetup*.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *nmtsetup* – the current number of calls to *mtsetup*.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Note:** This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is “attached” to ARKStep, or when ARKStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumMTSetups\(\)](#) instead.

int **ARKStepGetLastMassFlag**(void \*arkode\_mem, long int \*mlsflag)

Returns the last return value from an ARKLS mass matrix interface routine.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *mlsflag* – the value of the last return flag from an ARKLS mass matrix solver interface function.

**Return value:**

- *ARKLS\_SUCCESS* if successful
- *ARKLS\_MEM\_NULL* if the ARKStep memory was NULL
- *ARKLS\_LMEM\_NULL* if the linear solver memory was NULL

**Notes:**

The values of *msflag* for each of the various solvers will match those described above for the function [ARKStepGetLastLinFlag\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeGetLastMassFlag\(\)](#) instead.

**General usability functions**

int **ARKStepWriteParameters**(void \*arkode\_mem, FILE \*fp)

Outputs all ARKStep solver parameters to the provided file pointer.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *fp* – pointer to use for printing the solver parameters.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

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

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

Deprecated since version 6.1.0: Use [ARKodeWriteParameters\(\)](#) instead.

int **ARKStepWriteButcher**(void \*arkode\_mem, FILE \*fp)

Outputs the current Butcher table(s) to the provided file pointer.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *fp* – pointer to use for printing the Butcher table(s).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL

**Notes:**

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

If ARKStep is currently configured to run in purely explicit or purely implicit mode, this will output a single Butcher table; if configured to run an ImEx method then both tables will be output.

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

Deprecated since version 6.1.0: Use [ARKStepGetCurrentButcherTables\(\)](#) and [ARKodeButcherTable\\_Write\(\)](#) instead.

### 5.7.1.11 ARKStep re-initialization function

To reinitialize the ARKStep module for the solution of a new problem, where a prior call to `ARKStepCreate()` has been made, the user must call the function `ARKStepReInit()`. The new problem must have the same size as the previous one. This routine retains the current settings for all ARKStep module options and performs the same input checking and initializations that are done in `ARKStepCreate()`, but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration, and deletes any previously-set *tstop* value specified via a call to `ARKStepSetStopTime()`. Following a successful call to `ARKStepReInit()`, call `ARKStepEvolve()` again for the solution of the new problem.

The use of `ARKStepReInit()` requires that the number of Runge–Kutta stages, denoted by *s*, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the method order *q* and the problem type (explicit, implicit, ImEx) are left unchanged.

When using the ARKStep time-stepping module, if there are changes to the linear solver specifications, the user should make the appropriate calls to either the linear solver objects themselves, or to the ARKLS interface routines, as described in §5.7.1.3. Otherwise, all solver inputs set previously remain in effect.

One important use of the `ARKStepReInit()` function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to `ARKStepReInit()`. To stop when the location of the discontinuity is known, simply make that location a value of *tout*. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through *user\_data*) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **ARKStepReInit**(void \*arkode\_mem, *ARKRhsFn* fe, *ARKRhsFn* fi, *sunrealtype* t0, *N\_Vector* y0)

Provides required problem specifications and re-initializes the ARKStep time-stepper module.

#### Arguments:

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

#### Return value:

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_MEM\_FAIL* if a memory allocation failed
- *ARK\_ILL\_INPUT* if an argument had an illegal value.

#### Notes:

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, `ARKStepReInit()` also sends an error message to the error handler function.

### 5.7.1.12 ARKStep reset function

int **ARKStepReset**(void \*arkode\_mem, *sunrealtype* tR, *N\_Vector* yR)

Resets the current ARKStep time-stepper module state to the provided independent variable value and dependent variable vector.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *tR* – the value of the independent variable  $t$ .
- *yR* – the value of the dependent variable vector  $y(t_R)$ .

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_MEM\_FAIL* if a memory allocation failed
- *ARK\_ILL\_INPUT* if an argument had an illegal value.

**Notes:**

By default the next call to [ARKStepEvolve\(\)](#) will use the step size computed by ARKStep prior to calling [ARKStepReset\(\)](#). To set a different step size or have ARKStep estimate a new step size use [ARKStepSetInitStep\(\)](#).

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, [ARKStepReset\(\)](#) also sends an error message to the error handler function.

Deprecated since version 6.1.0: Use [ARKodeReset\(\)](#) instead.

### 5.7.1.13 ARKStep system resize function

int **ARKStepResize**(void \*arkode\_mem, *N\_Vector* yR, *sunrealtype* hscale, *sunrealtype* tR, *ARKVecResizeFn* resize, void \*resize\_data)

Re-sizes ARKStep with a different state vector but with comparable dynamical time scale.

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *yR* – the newly-sized state vector, holding the current dependent variable values  $y(t_R)$ .
- *hscale* – the desired time step scaling factor (i.e. the next step will be of size  $h*hscale$ ).
- *tR* – the current value of the independent variable  $t_R$  (this must be consistent with *yR*).
- *resize* – the user-supplied vector resize function (of type [ARKVecResizeFn\(\)](#)).
- *resize\_data* – the user-supplied data structure to be passed to *resize* when modifying internal ARKStep vectors.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ARKStep memory was NULL
- *ARK\_NO\_MALLOC* if *arkode\_mem* was not allocated.
- *ARK\_ILL\_INPUT* if an argument had an illegal value.



**Notes:**

If an error occurred, `ARKStepResize()` also sends an error message to the error handler function.

If inequality constraint checking is enabled a call to `ARKStepResize()` will disable constraint checking. A call to `ARKStepSetConstraints()` is required to re-enable constraint checking.

**Resizing the linear solver:**

When using any of the SUNDIALS-provided linear solver modules, the linear solver memory structures must also be resized. At present, none of these include a solver-specific “resize” function, so the linear solver memory must be destroyed and re-allocated **following** each call to `ARKStepResize()`. Moreover, the existing ARKLS interface should then be deleted and recreated by attaching the updated `SUNLinearSolver` (and possibly `SUNMatrix`) object(s) through calls to `ARKStepSetLinearSolver()`, and `ARKStepSetMassLinearSolver()`.

If any user-supplied routines are provided to aid the linear solver (e.g. Jacobian construction, Jacobian-vector product, mass-matrix-vector product, preconditioning), then the corresponding “set” routines must be called again **following** the solver re-specification.

**Resizing the absolute tolerance array:**

If using array-valued absolute tolerances, the absolute tolerance vector will be invalid after the call to `ARKStepResize()`, so the new absolute tolerance vector should be re-set **following** each call to `ARKStepResize()` through a new call to `ARKStepSVtolerances()` and possibly `ARKStepResVtolerance()` if applicable.

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

**Example codes:**

- `examples/arkode/C_serial/ark_heat1D_adapt.c`

Deprecated since version 6.1.0: Use `ARKodeResize()` instead.

**5.7.1.14 Interfacing with MRISStep**

When using ARKStep as the inner (fast) integrator with MRISStep, the utility function `ARKStepCreateMRISStepInnerStepper()` should be used to wrap an ARKStep memory block as an `MRISStepInnerStepper`.

int `ARKStepCreateMRISStepInnerStepper`(void \*inner\_arkode\_mem, `MRISStepInnerStepper` \*stepper)

Wraps an ARKStep memory block as an `MRISStepInnerStepper` for use with MRISStep.

**Arguments:**

- `arkode_mem` – pointer to the ARKStep memory block.
- `stepper` – the `MRISStepInnerStepper` object.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_FAIL` if a memory allocation failed
- `ARK_ILL_INPUT` if an argument had an illegal value.

**Example usage:**

```
/* fast (inner) and slow (outer) ARKODE objects */
void *inner_arkode_mem = NULL;
void *outer_arkode_mem = NULL;
```

(continues on next page)

(continued from previous page)

```

/* MRISStepInnerStepper to wrap the inner (fast) ARKStep object */
MRISStepInnerStepper stepper = NULL;

/* create an ARKStep object, setting fast (inner) right-hand side
   functions and the initial condition */
inner_arkode_mem = ARKStepCreate(fffe, ffi, t0, y0, sunctx);

/* setup ARKStep */
. . .

/* create MRISStepInnerStepper wrapper for the ARKStep memory block */
flag = ARKStepCreateMRISStepInnerStepper(inner_arkode_mem, &stepper);

/* create an MRISStep object, setting the slow (outer) right-hand side
   functions and the initial condition */
outer_arkode_mem = MRISStepCreate(fse, fsi, t0, y0, stepper, sunctx);

```

**Example codes:**

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

Deprecated since version 6.2.0: Use `ARKodeCreateMRISStepInnerStepper()` instead.

## 5.7.2 Relaxation Methods

This section describes ARKStep-specific user-callable functions for applying relaxation methods with ARKStep. All of these routines have been deprecated in favor of *shared ARKODE-level routines*, but this documentation will be retained for as long as these functions are present in the library.

### 5.7.2.1 Enabling or Disabling Relaxation

int **ARKStepSetRelaxFn**(void \*arkode\_mem, *ARKRelaxFn* rfn, *ARKRelaxJacFn* rjac)

Attaches the user supplied functions for evaluating the relaxation function (*rfn*) and its Jacobian (*rjac*).

Both *rfn* and *rjac* are required and an error will be returned if only one of the functions is NULL. If both *rfn* and *rjac* are NULL, relaxation is disabled.

With DIRK and IMEX-ARK methods or when a fixed mass matrix is present, applying relaxation requires allocating *s* additional state vectors (where *s* is the number of stages in the method).

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **rfn** – the user-defined function to compute the relaxation function  $\xi(y)$
- **rjac** – the user-defined function to compute the relaxation Jacobian  $\xi'(y)$

**Return values**

- **ARK\_SUCCESS** – the function exited successfully
- **ARK\_MEM\_NULL** – *arkode\_mem* was NULL
- **ARK\_ILL\_INPUT** – an invalid input combination was provided (see the output error message for more details)

- **ARK\_MEM\_FAIL** – a memory allocation failed

**Warning**

Applying relaxation requires using a method of at least second order with  $b_i^E \geq 0$  and  $b_i^I \geq 0$ . If these conditions are not satisfied, `ARKStepEvolve()` will return with an error during initialization.

**Note**

When combined with fixed time step sizes, ARKStep will attempt each step using the specified step size. If the step is successful, relaxation will be applied, effectively modifying the step size for the current step. If the step fails or applying relaxation fails, `ARKStepEvolve()` will return with an error.

Added in version 5.6.0.

Deprecated since version 6.1.0: Use `ARKodeSetRelaxFn()` instead.

### 5.7.2.2 Optional Input Functions

This section describes optional input functions used to control applying relaxation.

int **ARKStepSetRelaxEtaFail**(void \*arkode\_mem, *sunrealtype* eta\_rf)

Sets the step size reduction factor applied after a failed relaxation application.

The default value is 0.25. Input values  $\leq 0$  or  $\geq 1$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **eta\_rf** – the step size reduction factor

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use `ARKodeSetRelaxEtaFail()` instead.

int **ARKStepSetRelaxLowerBound**(void \*arkode\_mem, *sunrealtype* lower)

Sets the smallest acceptable value for the relaxation parameter.

Values smaller than the lower bound will result in a failed relaxation application and the step will be repeated with a smaller step size (determined by `ARKStepSetRelaxEtaFail()`).

The default value is 0.8. Input values  $\leq 0$  or  $\geq 1$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **lower** – the relaxation parameter lower bound

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxLowerBound\(\)](#) instead.

int **ARKStepSetRelaxUpperBound**(void \*arkode\_mem, *sunrealtype* upper)

Sets the largest acceptable value for the relaxation parameter.

Values larger than the upper bound will result in a failed relaxation application and the step will be repeated with a smaller step size (determined by [ARKStepSetRelaxEtaFail\(\)](#)).

The default value is 1.2. Input values  $\leq 1$  will result in the default value being used.

#### Parameters

- **arkode\_mem** – the ARKStep memory structure
- **upper** – the relaxation parameter upper bound

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxUpperBound\(\)](#) instead.

int **ARKStepSetRelaxMaxFails**(void \*arkode\_mem, int max\_fails)

Sets the maximum number of times applying relaxation can fail within a step attempt before the integration is halted with an error.

The default value is 10. Input values  $\leq 0$  will result in the default value being used.

#### Parameters

- **arkode\_mem** – the ARKStep memory structure
- **max\_fails** – the maximum number of failed relaxation applications allowed in a step

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxMaxFails\(\)](#) instead.

int **ARKStepSetRelaxMaxIters**(void \*arkode\_mem, int max\_iters)

Sets the maximum number of nonlinear iterations allowed when solving for the relaxation parameter.

If the maximum number of iterations is reached before meeting the solve tolerance (determined by [ARKStepSetRelaxResTol\(\)](#) and [ARKStepSetRelaxTol\(\)](#)), the step will be repeated with a smaller step size (determined by [ARKStepSetRelaxEtaFail\(\)](#)).

The default value is 10. Input values  $\leq 0$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **max\_iters** – the maximum number of solver iterations allowed

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxMaxIters\(\)](#) instead.

int **ARKStepSetRelaxSolver**(void \*arkode\_mem, *ARKRelaxSolver* solver)

Sets the nonlinear solver method used to compute the relaxation parameter.

The default value is [ARK\\_RELAX\\_NEWTON](#)

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **solver** – the nonlinear solver to use

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL
- **ARK\_ILL\_INPUT** – an invalid solver option was provided

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxSolver\(\)](#) instead.

int **ARKStepSetRelaxResTol**(void \*arkode\_mem, *sunrealtype* res\_tol)

Sets the nonlinear solver residual tolerance to use when solving (2.63).

If the residual or iteration update tolerance (see [ARKStepSetRelaxMaxIters\(\)](#)) is not reached within the maximum number of iterations (determined by [ARKStepSetRelaxMaxIters\(\)](#)), the step will be repeated with a smaller step size (determined by [ARKStepSetRelaxEtaFail\(\)](#)).

The default value is  $4\epsilon$  where  $\epsilon$  is floating-point precision. Input values  $\leq 0$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **res\_tol** – the nonlinear solver residual tolerance to use

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxResTol\(\)](#) instead.

int **ARKStepSetRelaxTol**(void \*arkode\_mem, *sunrealtype* rel\_tol, *sunrealtype* abs\_tol)

Sets the nonlinear solver relative and absolute tolerance on changes in  $r$  iterates when solving (2.63).

If the residual (see [ARKStepSetRelaxResTol\(\)](#)) or iterate update tolerance is not reached within the maximum number of iterations (determined by [ARKStepSetRelaxMaxIters\(\)](#)), the step will be repeated with a smaller step size (determined by [ARKStepSetRelaxEtaFail\(\)](#)).

The default relative and absolute tolerances are  $4\epsilon$  and  $10^{-14}$ , respectively, where  $\epsilon$  is floating-point precision. Input values  $\leq 0$  will result in the default value being used.

#### Parameters

- **arkode\_mem** – the ARKStep memory structure
- **rel\_tol** – the nonlinear solver relative solution tolerance to use
- **abs\_tol** – the nonlinear solver absolute solution tolerance to use

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxTol\(\)](#) instead.

### 5.7.2.3 Optional Output Functions

This section describes optional output functions used to retrieve information about the performance of the relaxation method.

int **ARKStepGetNumRelaxFnEvals**(void \*arkode\_mem, long int \*r\_evals)

Get the number of times the user's relaxation function was evaluated.

#### Parameters

- **arkode\_mem** – the ARKStep memory structure
- **r\_evals** – the number of relaxation function evaluations

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxFnEvals\(\)](#) instead.

int **ARKStepGetNumRelaxJacEvals**(void \*arkode\_mem, long int \*J\_evals)

Get the number of times the user's relaxation Jacobian was evaluated.

#### Parameters

- **arkode\_mem** – the ARKStep memory structure

- **J\_evals** – the number of relaxation Jacobian evaluations

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxJacEvals\(\)](#) instead.

int **ARKStepGetNumRelaxFails**(void \*arkode\_mem, long int \*fails)

Get the total number of times applying relaxation failed.

The counter includes the sum of the number of nonlinear solver failures (see [ARKStepGetNumRelaxSolveFails\(\)](#)) and the number of failures due an unacceptable relaxation value (see [ARKStepSetRelaxLowerBound\(\)](#) and [ARKStepSetRelaxUpperBound\(\)](#)).

#### Parameters

- **arkode\_mem** – the ARKStep memory structure
- **fails** – the total number of failed relaxation attempts

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxFails\(\)](#) instead.

int **ARKStepGetNumRelaxBoundFails**(void \*arkode\_mem, long int \*fails)

Get the number of times the relaxation parameter was deemed unacceptable.

#### Parameters

- **arkode\_mem** – the ARKStep memory structure
- **fails** – the number of failures due to an unacceptable relaxation parameter value

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxBoundFails\(\)](#) instead.

int **ARKStepGetNumRelaxSolveFails**(void \*arkode\_mem, long int \*fails)

Get the number of times the relaxation parameter nonlinear solver failed.

#### Parameters

- **arkode\_mem** – the ARKStep memory structure
- **fails** – the number of relaxation nonlinear solver failures

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [`ARKodeGetNumRelaxSolveFails\(\)`](#) instead.

int **ARKStepGetNumRelaxSolveIters**(void \*arkode\_mem, long int \*iters)

Get the number of relaxation parameter nonlinear solver iterations.

**Parameters**

- **arkode\_mem** – the ARKStep memory structure
- **iters** – the number of relaxation nonlinear solver iterations

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – `arkode_mem` was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [`ARKodeGetNumRelaxSolveIters\(\)`](#) instead.

### 5.7.3 Multigrid Reduction in Time with XBraid

The prior sections discuss using ARKODE in a traditional sequential time integration setting i.e., the solution is advanced from one time to the next where all parallelism resides within the evaluation of a step e.g., the computation of the right-hand side, (non)linear solves, vector operations etc. For example, when discretizing a partial differential equation using a method of lines approach the spatially-discretized equations comprise a large set of ordinary differential equations that can be evolved with ARKODE. In this case the parallelization lies in decomposing the spatial domain unknowns across distributed computational nodes. Considering the strong scaling case at a given spatial resolution, as the problem is spread across greater numbers of computational nodes scalability in the spatial dimension is exhausted and sequential time integration becomes a bottleneck. This bottleneck is largely driven by the hardware shift from faster clock speeds to greater concurrency to achieve performance gains. In this case, at the spatial scaling limit and with stagnant clock speeds, more time steps will lead to an increased runtime.

An alternative approach to sequential time integration is to solve for all time values simultaneously. One such approach is multigrid reduction in time [41] (MGRIT) which uses a highly parallel iterative method to expose parallelism in the time domain in addition to the spatial parallelization. Starting with an initial temporal grid the multilevel algorithm constructs successively coarser time grids and uses each coarse grid solution to improve the solution at the next finer scale. In the two level case the MGRIT algorithm is as follows:

1. Relax the solution on the fine grid (parallel-in-time)
2. Restrict the solution to the fine grid (time re-discretization).
3. Solve the residual equation on the coarse grid (serial-in-time).
4. Correct the fine grid solution (parallel-in-time).

Applying this algorithm recursively for the solve step above leads to the multilevel algorithm.

The XBraid library [1] implements the MGRIT algorithm in a non-intrusive manner, enabling the reuse of existing software for sequential time integration. The following sections describe the ARKODE + XBraid interface and the



steps necessary to modify an existing code that already uses ARKODE's ARKStep time-stepping module to also use XBraid.

### 5.7.3.1 SUNBraid Interface

Interfacing ARKODE with XBraid requires defining two data structures. The first is the XBraid application data structure that contains the data necessary for carrying out a time step and is passed to every interface function (much like the user data pointer in SUNDIALS packages). For this structure the SUNBraid interface defines the generic SUNBraidApp structure described below that serves as the basis for creating integrator-specific or user-defined interfaces to XBraid. The second structure holds the problem state data at a certain time value. This structure is defined by the SUNBraidVector structure and simply contains an N\_Vector. In addition to the two data structures several functions defined by the XBraid API are required. These functions include vector operations (e.g., computing vector sums or norms) as well as functions to initialize the problem state, access the current solution, and take a time step.

The ARKBraid interface, built on the SUNBraidApp and SUNBraidVector structures, provides all the functionality needed combine ARKODE and XBraid for parallel-in-time integration. As such, only a minimal number of changes are necessary to update an existing code that uses ARKODE to also use XBraid.

#### SUNBraidApp

As mentioned above the SUNBraid interface defines the SUNBraidApp structure to hold the data necessary to compute a time step. This structure, like other SUNDIALS generic objects, is defined as a structure consisting of an implementation specific *content* field and an operations structure comprised of a set of function pointers for implementation-defined operations on the object. Specifically the SUNBraidApp type is defined as

```
/* Define XBraid App structure */
struct _braid_App_struct
{
    void          *content;
    SUNBraidOps ops;
};

/* Pointer to the interface object (same as braid_App) */
typedef struct _braid_App_struct *SUNBraidApp;
```

Here, the SUNBraidOps structure is defined as

```
/* Structure containing function pointers to operations */
struct _SUNBraidOps
{
    int (*getvectmpl)(braid_App app, N_Vector *tmpl);
};

/* Pointer to operations structure */
typedef struct _SUNBraidOps *SUNBraidOps;
```

The generic SUNBraidApp defines and implements the generic operations acting on a SUNBraidApp object. These generic functions are nothing but wrappers to access the specific implementation through the object's operations structure. To illustrate this point we show below the implementation of the `SUNBraidApp_GetVecTpl()` function:

```
/* Get a template vector from the integrator */
int SUNBraidApp_GetVecTpl(braid_App app, N_Vector *y)
{
```

(continues on next page)

(continued from previous page)

```
if (app->ops->getvectmpl == NULL) return SUNBRAID_OPNULL;
return app->ops->getvectmpl(app, y);
}
```

The SUNBraidApp operations are define below in §5.7.3.1.

## SUNBraidOps

In this section we define the SUNBraidApp operations and, for each operation, we give the function signature, a description of the expected behavior, and an example usage of the function.

int **SUNBraidApp\_GetVecTpl**(braid\_App app, *N\_Vector* \*y)

This function returns a vector to use as a template for creating new vectors with *N\_VClone()*.

### Parameters

- **app** – input, a SUNBraidApp instance (XBraid app structure).
- **y** – output, the template vector.

### Returns

If this function is not implemented by the SUNBraidApp implementation (i.e., the function pointer is NULL) then this function will return *SUNBRAID\_OPNULL*. Otherwise the return value depends on the particular SUNBraidApp implementation. Users are encouraged to utilize the return codes defined in *sundials/sundials\_xbraid.h* and listed in Table 5.2.

```
/* Get template vector */
flag = SUNBraidApp_GetVecTpl(app, y_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

## SUNBraidApp Utility Functions

In addition to the generic SUNBraidApp operations the following utility functions are provided to assist in creating and destroying a SUNBraidApp instance.

int **SUNBraidApp\_NewEmpty**(braid\_App \*app)

This function creates a new SUNBraidApp instance with the content and operations initialized to NULL.

### Parameters

- **app** – output, an empty SUNBraidApp instance (XBraid app structure).

### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ALLOCFAIL** – if a memory allocation failed.

```
/* Create empty XBraid interface object */
flag = SUNBraidApp_NewEmpty(app_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

int **SUNBraidApp\_FreeEmpty**(braid\_App \*app)

This function destroys an empty SUNBraidApp instance.

### Parameters

- **app** – input, an empty SUNBraidApp instance (XBraid app structure).

**Return values**

**SUNBRAID\_SUCCESS** – if successful.

```
/* Free empty XBraid interface object */
flag = SUNBraidApp_FreeEmpty(app_ptr);
```

**Warning**

This function does not free the SUNBraidApp object's content structure. An implementation should free its content before calling *SUNBraidApp\_FreeEmpty()* to deallocate the base SUNBraidApp structure.

**SUNBraidVector**

As mentioned above the SUNBraid interface defines the SUNBraidVector structure to store a snapshot of solution data at a single point in time and this structure simply contains an N\_Vector. Specifically, the structure is defined as follows:

```
typedef struct _braid_Vector_struct *SUNBraidVector;
```

Pointer to vector wrapper (same as *braid\_Vector*)

```
struct _braid_Vector_struct
```

*N\_Vector* **y**

SUNDIALS N\_Vector wrapped by the *braid\_Vector*

To assist in creating and destroying this structure the following utility functions are provided.

```
int SUNBraidVector_New(N_Vector y, SUNBraidVector *u)
```

This function creates a new SUNBraidVector wrapping the N\_Vector y.

**Parameters**

- **y** – input, the N\_Vector to wrap.
- **u** – output, the SUNBraidVector wrapping y.

**Return values**

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if y is NULL.
- **SUNBRAID\_ALLOCFAIL** – if a memory allocation fails.

```
/* Create new vector wrapper */
flag = SUNBraidVector_New(y, u_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

**Warning**

The SUNBraidVector takes ownership of the wrapped N\_Vector and as such the wrapped N\_Vector is destroyed when the SUNBraidVector is freed with *SUNBraidVector\_Free()*.

int **SUNBraidVector\_GetNVector**(*SUNBraidVector* u, *N\_Vector* \*y)

This function retrieves the wrapped *N\_Vector* from the *SUNBraidVector*.

**Parameters**

- **u** – input, the *SUNBraidVector* wrapping *y*.
- **y** – output, the wrapped *N\_Vector*.

**Return values**

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *u* is NULL.
- **SUNBRAID\_MEMFAIL** – if *y* is NULL.

```
/* Create new vector wrapper */
flag = SUNBraidVector_GetNVector(u, y_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

Finally, the *SUNBraid* interface defines the following vector operations acting on *SUNBraidVectors*, that consist of thin wrappers to compatible SUNDIALS *N\_Vector* operations.

int **SUNBraidVector\_Clone**(*braid\_App* app, *braid\_Vector* u, *braid\_Vector* \*v\_ptr)

This function creates a clone of the input *SUNBraidVector* and copies the values of the input vector *u* into the output vector *v\_ptr* using *N\_VClone()* and *N\_VScale()*.

**Parameters**

- **app** – input, a *SUNBraidApp* instance (XBraid app structure).
- **u** – input, the *SUNBraidVector* to clone.
- **v\_ptr** – output, the new *SUNBraidVector*.

**Return values**

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *u* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *N\_Vector* *y* wrapped by *u* is NULL.
- **SUNBRAID\_ALLOCFAIL** – if a memory allocation fails.

int **SUNBraidVector\_Free**(*braid\_App* app, *braid\_Vector* u)

This function destroys the *SUNBraidVector* and the wrapped *N\_Vector* using *N\_VDestroy()*.

**Parameters**

- **app** – input, a *SUNBraidApp* instance (XBraid app structure).
- **u** – input, the *SUNBraidVector* to destroy.

**Return values**

**SUNBRAID\_SUCCESS** – if successful.

int **SUNBraidVector\_Sum**(*braid\_App* app, *braid\_Real* alpha, *braid\_Vector* x, *braid\_Real* beta, *braid\_Vector* y)

This function computes the vector sum  $\alpha x + \beta y \rightarrow y$  using *N\_VLinearSum()*.

**Parameters**

- **app** – input, a *SUNBraidApp* instance (XBraid app structure).
- **alpha** – input, the constant  $\alpha$ .

- **x** – input, the vector  $x$ .
- **beta** – input, the constant  $\beta$ .
- **y** – input/output, the vector  $y$ .

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if  $x$  or  $y$  is NULL.
- **SUNBRAID\_MEMFAIL** – if either of the wrapped `N_Vectors` are NULL.

int **SUNBraidVector\_SpatialNorm**(braid\_App app, braid\_Vector u, braid\_Real \*norm\_ptr)

This function computes the 2-norm of the vector  $u$  using [N\\_VDotProd\(\)](#).

#### Parameters

- **app** – input, a SUNBraidApp instance (XBraid app structure).
- **u** – input, the vector  $u$ .
- **norm\_ptr** – output, the L2 norm of  $u$ .

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if  $u$  is NULL.
- **SUNBRAID\_MEMFAIL** – if the wrapped `N_Vector` is NULL.

int **SUNBraidVector\_BufSize**(braid\_App app, braid\_Int \*size\_ptr, braid\_BufferStatus bstatus)

This function returns the buffer size for messages to exchange vector data using [SUNBraidApp\\_GetVecTpl\(\)](#) and [N\\_VBufSize\(\)](#).

#### Parameters

- **app** – input, a SUNBraidApp instance (XBraid app structure).
- **size\_ptr** – output, the buffer size.
- **bstatus** – input, a status object to query for information on the message type.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful
- **otherwise** – an error flag from [SUNBraidApp\\_GetVecTpl\(\)](#) or [N\\_VBufSize\(\)](#).

int **SUNBraidVector\_BufPack**(braid\_App app, braid\_Vector u, void \*buffer, braid\_BufferStatus bstatus)

This function packs the message buffer for exchanging vector data using [N\\_VBufPack\(\)](#).

#### Parameters

- **app** – input, a SUNBraidApp instance (XBraid app structure).
- **u** – input, the vector to pack into the exchange buffer.
- **buffer** – output, the packed exchange buffer to pack.
- **bstatus** – input, a status object to query for information on the message type.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if  $u$  is NULL.

- **otherwise** – An error flag from `N_VBufPack()`.

int **SUNBraidVector\_BufUnpack**(braid\_App app, void \*buffer, braid\_Vector \*u\_ptr, braid\_BufferStatus bstatus)

This function unpacks the message buffer and creates a new `N_Vector` and `SUNBraidVector` with the buffer data using `N_VBufUnpack()`, `SUNBraidApp_GetVecTmp1()`, and `N_VClone()`.

#### Parameters

- **app** – input, a `SUNBraidApp` instance (XBraid app structure).
- **buffer** – input, the exchange buffer to unpack.
- **u\_ptr** – output, a new `SUNBraidVector` containing the buffer data.
- **bstatus** – input, a status object to query for information on the message type.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *buffer* is NULL.
- **SUNBRAID\_ALLOCFAIL** – if a memory allocation fails.
- **otherwise** – an error flag from `SUNBraidApp_GetVecTmp1()` and `N_VBufUnpack()`.

### SUNBraid Return Codes

The SUNBraid interface return values are given in [Table 5.2](#).

Table 5.2: SUNBraid Return Codes

Return value name	Value	Meaning
SUNBRAID_SUCCESS	0	The call/operation was successful.
SUNBRAID_ALLOCFAIL	-1	A memory allocation failed.
SUNBRAID_MEMFAIL	-2	A memory access fail.
SUNBRAID_OPNULL	-3	The SUNBraid operation is NULL.
SUNBRAID_ILLINPUT	-4	An invalid input was provided.
SUNBRAID_BRAIDFAIL	-5	An XBraid function failed.
SUNBRAID_SUNFAIL	-6	A SUNDIALS function failed.

#### 5.7.3.2 ARKBraid Interface

This section describes the ARKBraid implementation of a `SUNBraidApp` for using the ARKODE's ARKStep time-stepping module with XBraid. The following section §5.7.3.2 describes routines for creating, initializing, and destroying the ARKODE + XBraid interface, routines for setting optional inputs, and routines for retrieving data from an ARKBraid instance. As noted above, interfacing with XBraid requires providing functions to initialize the problem state, access the current solution, and take a time step. The default ARKBraid functions for each of these actions are defined in §5.7.3.2 and may be overridden by user-defined if desired. A skeleton of the user's main or calling program for using the ARKBraid interface is given in §5.7.3.3. Finally, for advanced users that wish to create their own `SUNBraidApp` implementation using ARKODE, §5.7.3.4 describes some helpful functions available to the user.

## ARKBraid Initialization and Deallocation Functions

This section describes the functions that are called by the user to create, initialize, and destroy an ARKBraid instance. Each user-callable function returns `SUNBRAID_SUCCESS` (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in [Table 5.2](#).

int **ARKBraid\_Create**(void \*arkode\_mem, braid\_App \*app)

This function creates a SUNBraidApp object, sets the content pointer to the private ARKBraid interface structure, and attaches the necessary SUNBraidOps implementations.

### Parameters

- **arkode\_mem** – input, a pointer to an ARKODE memory structure.
- **app** – output, an ARKBraid instance (XBraid app structure).

### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – *arkode\_mem* is NULL.
- **SUNBRAID\_ALLOCFAIL** – if a memory allocation failed.

### Warning

The ARKBraid interface is ARKStep-specific. Although one could eventually construct an XBraid interface to other of ARKODE time-stepping modules (e.g., ERKStep or MRISStep), those are not currently supported by this implementation.

int **ARKBraid\_BraidInit**(MPI\_Comm comm\_w, MPI\_Comm comm\_t, *sunrealtype* tstart, *sunrealtype* tstop, *sunindextype* ntime, braid\_App app, braid\_Core \*core)

This function wraps the XBraid `braid_Init()` function to create the XBraid core memory structure and initializes XBraid with the ARKBraid and SUNBraidVector interface functions.

### Parameters

- **comm\_w** – input, the global MPI communicator for space and time.
- **comm\_t** – input, the MPI communicator for the time dimension.
- **tstart** – input, the initial time value.
- **tstop** – input, the final time value.
- **ntime** – input, the initial number of grid points in time.
- **app** – input, an ARKBraid instance.
- **core** – output, the XBraid core memory structure.

### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if either MPI communicator is `MPI_COMM_NULL`, if *ntime* < 2, or if *app* or its content is NULL.
- **SUNBRAID\_BRAIDFAIL** – if the `braid_Init()` call fails. The XBraid return value can be retrieved with [ARKBraid\\_GetLastBraidFlag\(\)](#).

**Note**

If desired, the default functions for vector initialization, accessing the solution, taking a time step, and computing the spatial norm should be overridden before calling this function. See §5.7.3.2 for more details.

**Warning**

The user is responsible for deallocating the XBraid core memory structure with the XBraid function `braid_Destroy()`.

int **ARKBraid\_Free**(braid\_App \*app)

This function deallocates an ARKBraid instance.

**Parameters**

- **app** – input, a pointer to an ARKBraid instance.

**Return values**

**SUNBRAID\_SUCCESS** – if successful.

**ARKBraid Set Functions**

This section describes the functions that are called by the user to set optional inputs to control the behavior of an ARKBraid instance or to provide alternative XBraid interface functions. Each user-callable function returns **SUNBRAID\_SUCCESS** (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in Table 5.2.

int **ARKBraid\_SetStepFn**(braid\_App app, braid\_PtFcnStep step)

This function sets the step function provided to XBraid (default [\*ARKBraid\\_Step\(\)\*](#)).

**Parameters**

- **app** – input, an ARKBraid instance.
- **step** – input, an XBraid step function. If *step* is NULL, the default function will be used.

**Return values**

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content is NULL.

**Note**

This function must be called prior to [\*ARKBraid\\_BraidInit\(\)\*](#).

int **ARKBraid\_SetInitFn**(braid\_App app, braid\_PtFcnInit init)

This function sets the vector initialization function provided to XBraid (default [\*ARKBraid\\_Init\(\)\*](#)).

**Parameters**

- **app** – input, an ARKBraid instance.



- **init** – input, an XBraid vector initialization function. If *init* is NULL, the default function will be used.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content is NULL.

#### Note

This function must be called prior to [ARKBraid\\_BraidInit\(\)](#).

int **ARKBraid\_SetSpatialNormFn**(braid\_App app, braid\_PtFcnSpatialNorm snorm)

This function sets the spatial norm function provided to XBraid (default [SUNBraidVector\\_SpatialNorm\(\)](#)).

#### Parameters

- **app** – input, an ARKBraid instance.
- **snorm** – input, an XBraid spatial norm function. If *snorm* is NULL, the default function will be used.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content is NULL.

#### Note

This function must be called prior to [ARKBraid\\_BraidInit\(\)](#).

int **ARKBraid\_SetAccessFn**(braid\_App app, braid\_PtFcnAccess access)

This function sets the user access function provided to XBraid (default [ARKBraid\\_Access\(\)](#)).

#### Parameters

- **app** – input, an ARKBraid instance.
- **init** – input, an XBraid user access function. If *access* is NULL, the default function will be used.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content is NULL.

#### Note

This function must be called prior to [ARKBraid\\_BraidInit\(\)](#).

## ARKBraid Get Functions

This section describes the functions that are called by the user to retrieve data from an ARKBraid instance. Each user-callable function returns `SUNBRAID_SUCCESS` (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in [Table 5.2](#).

int **ARKBraid\_GetVecTpl**(braid\_App app, *N\_Vector* \*tpl)

This function returns a vector from the ARKODE memory to use as a template for creating new vectors with [`N\_VClone\(\)`](#) i.e., this is the ARKBraid implementation of [`SUNBraidApp\_GetVecTpl\(\)`](#).

### Parameters

- **app** – input, an ARKBraid instance.
- **tpl** – output, a template vector.

### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content or ARKODE memory is NULL.

int **ARKBraid\_GetARKodeMem**(braid\_App app, void \*\*arkode\_mem)

This function returns the ARKODE memory structure pointer attached with [`ARKBraid\_Create\(\)`](#).

### Parameters

- **app** – input, an ARKBraid instance.
- **arkode\_mem** – output, a pointer to the ARKODE memory structure.

### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content or ARKODE memory is NULL.

int **ARKBraid\_GetARKStepMem**(braid\_App app, void \*\*arkode\_mem)

This function returns the ARKStep memory structure pointer attached with [`ARKBraid\_Create\(\)`](#).

### Parameters

- **app** – input, an ARKBraid instance.
- **arkode\_mem** – output, a pointer to the ARKStep memory structure.

### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content or ARKStep memory is NULL.

Deprecated since version 6.1.0: Use [`ARKBraid\_GetARKodeMem\(\)`](#) instead.

int **ARKBraid\_GetUserData**(braid\_App app, void \*\*user\_data)

This function returns the user data pointer attached with [`ARKodeSetUserData\(\)`](#).

### Parameters

- **app** – input, an ARKBraid instance.

- **user\_data** – output, a pointer to the user data structure.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content or ARKODE memory is NULL.

int **ARKBraid\_GetLastBraidFlag**(braid\_App app, int \*last\_flag)

This function returns the return value from the most recent XBraid function call.

#### Parameters

- **app** – input, an ARKBraid instance.
- **last\_flag** – output, the XBraid return value.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content is NULL.

int **ARKBraid\_GetLastARKodeFlag**(braid\_App app, int \*last\_flag)

This function returns the return value from the most recent ARKODE function call.

#### Parameters

- **app** – input, an ARKBraid instance.
- **last\_flag** – output, the ARKODE return value.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content is NULL.

int **ARKBraid\_GetLastARKStepFlag**(braid\_App app, int \*last\_flag)

This function returns the return value from the most recent ARKStep function call.

#### Parameters

- **app** – input, an ARKBraid instance.
- **last\_flag** – output, the ARKStep return value.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content is NULL.

Deprecated since version 6.1.0: Use [ARKBraid\\_GetLastARKodeFlag\(\)](#) instead.

int **ARKBraid\_GetSolution**(braid\_App app, *sunrealtype* \*tout, *N\_Vector* yout)

This function returns final time and state stored with the default access function [ARKBraid\\_Access\(\)](#).

#### Parameters

- **app** – input, an ARKBraid instance.

- **last\_flag** – output, the ARKODE return value.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content or the stored vector is NULL.

#### Warning

If providing a non-default access function the final time and state are not stored within the ARKBraid structure and this function will return an error. In this case the user should allocate space to store any desired output within the user data pointer attached to ARKODE with [ARKodeSetUserData\(\)](#). This user data pointer can be retrieved from the ARKBraid structure with [ARKBraid\\_GetUserData\(\)](#).

### ARKBraid Interface Functions

This section describes the default XBraid interface functions provided by ARKBraid and called by XBraid to perform certain actions. Any or all of these functions may be overridden by supplying a user-defined function through the set functions defined in §5.7.3.2. Each default interface function returns **SUNBRAID\_SUCCESS** (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in [Table 5.2](#).

int **ARKBraid\_Step**(braid\_App app, braid\_Vector ustop, braid\_Vector fstop, braid\_Vector u, braid\_StepStatus status)

This is the default step function provided to XBraid. The step function is called by XBraid to advance the vector *u* from one time to the next using the ARStep memory structure provided to [ARKBraid\\_Create\(\)](#). A user-defined step function may be set with [ARKBraid\\_SetStepFn\(\)](#).

#### Parameters

- **app** – input, an ARKBraid instance.
- **ustop** – input, *u* vector at the new time *tstop*.
- **fstop** – input, the right-hand side vector at the new time *tstop*.
- **u** – input/output, on input the vector at the start time and on return the vector at the new time.
- **status** – input, a status object to query for information about *u* and to steer XBraid e.g., for temporal refinement.

#### Return values

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content or ARKODE memory is NULL.
- **SUNBRAID\_BRAIDFAIL** – if an XBraid function fails. The return value can be retrieved with [ARKBraid\\_GetLastBraidFlag\(\)](#).
- **SUNBRAID\_SUNFAIL** – if a SUNDIALS function fails. The return value can be retrieved with [ARKBraid\\_GetLastARKStepFlag\(\)](#).

**Note**

If providing a non-default implementation of the step function the utility function [ARKBraid\\_TakeStep\(\)](#) should be used to advance the input vector  $u$  to the new time.

int **ARKBraid\_Init**(braid\_App app, *sunrealtype* t, braid\_Vector \*u\_ptr)

This is the default vector initialization function provided to XBraid. The initialization function is called by XBraid to create a new vector and set the initial guess for the solution at time  $t$ . When using this default function the initial guess at all time values is the initial condition provided to [ARKStepCreate\(\)](#). A user-defined init function may be set with [ARKBraid\\_SetInitFn\(\)](#).

**Parameters**

- **app** – input, an ARKBraid instance.
- **t** – input, the initialization time for the output vector.
- **u\_ptr** – output, the new and initialized SUNBraidVector.

**Return values**

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if *app* is NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content or ARKODE memory is NULL.
- **SUNBRAID\_ALLOCFAIL** – if a memory allocation failed.

**Note**

If providing a non-default implementation of the vector initialization function the utility functions [SUNBraidApp\\_GetVecTpl\(\)](#) and [SUNBraidVector\\_New\(\)](#) can be helpful when creating the new vector returned by this function.

int **ARKBraid\_Access**(braid\_App app, braid\_Vector u, braid\_AccessStatus astatus)

This is the default access function provided to XBraid. The access function is called by XBraid to retrieve the current solution. When using this default function the final solution time and state are stored within the ARKBraid structure. This information can be retrieved with [ARKBraid\\_GetSolution\(\)](#). A user-defined access function may be set with [ARKBraid\\_SetAccessFn\(\)](#).

**Parameters**

- **app** – input, an ARKBraid instance.
- **u** – input, the vector to be accessed.
- **status** – input, a status object to query for information about  $u$ .

**Return values**

- **SUNBRAID\_SUCCESS** – if successful.
- **SUNBRAID\_ILLINPUT** – if any of the inputs are NULL.
- **SUNBRAID\_MEMFAIL** – if the *app* content, the wrapped N\_Vector, or the ARKODE memory is NULL.
- **SUNBRAID\_ALLOCFAIL** – if allocating storage for the final solution fails.

- **SUNBRAID\_BRAIDFAIL** – if an XBraid function fails. The return value can be retrieved with [ARKBraid\\_GetLastBraidFlag\(\)](#).

### 5.7.3.3 A skeleton of the user's main program with XBraid

In addition to the header files required for the integration of the ODE problem (see the section §5.1), to use the ARK-Braid interface, the user's program must include the header file `arkode/arkode_xbraid.h` which declares the needed function prototypes.

The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP using ARKODE's ARKStep time-stepping module with XBraid for parallel-in-time integration. Most steps are unchanged from the skeleton program presented in §5.2. New or updated steps are **bold**.

1. **Initialize MPI**

If parallelizing in space and time split the global communicator into communicators for space and time with `braid_SplitCommworld()`.

2. *Set problem dimensions*

3. *Set vector of initial values*

4. *Create ARKStep object*

5. *Specify integration tolerances*

6. *Create matrix object*

7. *Create linear solver object*

8. *Set linear solver optional inputs*

9. *Attach linear solver module*

10. *Create nonlinear solver object*

11. *Attach nonlinear solver module*

12. *Set nonlinear solver optional inputs*

13. *Set optional inputs*

14. **Create ARKBraid interface**

Call the constructor [ARKBraid\\_Create\(\)](#) to create the XBraid app structure.

15. **Set optional ARKBraid inputs**

See §5.7.3.2 for ARKBraid inputs.

16. **Initialize the ARKBraid interface**

Call the initialization function [ARKBraid\\_BraidInit\(\)](#) to create the XBraid core memory structure and attach the ARKBraid interface app and functions.

17. **Set optional XBraid inputs**

See the XBraid documentation for available XBraid options.

18. **Evolve the problem**

Call `braid_Drive()` to evolve the problem with MGRIT.

19. **Get optional outputs**

See §5.7.3.2 for ARKBraid outputs.

20. *Deallocate memory for solution vector*

21. *Free solver memory*

22. *Free linear solver memory*

23. **Free ARKBraid and XBraid memory**

Call `ARKBraid_Free()` and `braid_Destroy` to deallocate the ARKBraid interface and and XBraid core memory structures, respectively.

24. *Finalize MPI*

### 5.7.3.4 Advanced ARKBraid Utility Functions

This section describes utility functions utilized in the ARKODE + XBraid interfacing. These functions are used internally by the above ARKBraid interface functions but are exposed to the user to assist in advanced usage of ARKODE and XBraid that requires defining a custom SUNBraidApp implementation.

int **ARKBraid\_TakeStep**(void \*arkode\_mem, *sunrealtype* tstart, *sunrealtype* tstop, *N\_Vector* y, int \*ark\_flag)

This function advances the vector *y* from *tstart* to *tstop* using a single ARKODE time step with step size  $h = tstop - tstart$ .

#### Parameters

- **arkode\_mem** – input, the ARKODE memory structure pointer.
- **tstart** – input, the step start time.
- **tstop** – input, the step stop time.
- **y** – input/output, on input the solution at *tstop* and on return, the solution at time *tstop* if the step was successful (*ark\_flag*  $\geq 0$ ) or the solution at time *tstart* if the step failed (*ark\_flag*  $< 0$ ).
- **ark\_flag** – output, the step status flag. If *ark\_flag* is:
  - = 0 then the step succeeded and, if applicable, met the requested temporal accuracy.
  - > 0 then the step succeeded but failed to meet the requested temporal accuracy.
  - < 0 then the step failed e.g., a solver failure occurred.

#### Returns

If all ARKODE function calls are successful the return value is `ARK_SUCCESS`, otherwise the return value is the error flag returned from the function that failed.

## 5.8 Using the ERKStep time-stepping module

This section is concerned with the use of the ERKStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. Usage of ERKStep follows that of the rest of ARKODE, and so in this section we primarily focus on those usage aspects that are specific to ERKStep.

### 5.8.1 ERKStep User-callable functions

This section describes the ERKStep-specific functions that may be called by the user to setup and then solve an IVP using the ERKStep time-stepping module. The large majority of these routines merely wrap *underlying ARKODE*

*functions*, and are now deprecated – each of these are clearly marked. However, some of these user-callable functions are specific to ERKStep, as explained below.

As discussed in the main [ARKODE user-callable function introduction](#), each of ARKODE’s time-stepping modules clarifies the categories of user-callable functions that it supports. ERKStep supports the following categories:

- temporal adaptivity
- relaxation Runge–Kutta methods

ERKStep also has forcing function support when converted to a [SUNStepper](#) or [MRISStepInnerStepper](#). See [ARKodeCreateSUNStepper\(\)](#) and [ARKStepCreateMRISStepInnerStepper\(\)](#) for additional details.

### 5.8.1.1 ERKStep initialization and deallocation functions

void **ERKStepCreate**([ARKRhsFn](#) f, [sunrealtype](#) t0, [N\\_Vector](#) y0, [SUNContext](#) sunctx)

This function allocates and initializes memory for a problem to be solved using the ERKStep time-stepping module in ARKODE.

**Arguments:**

- *f* – the name of the C function (of type [ARKRhsFn\(\)](#)) defining the right-hand side function in  $\dot{y} = f(t, y)$ .
- *t0* – the initial value of *t*.
- *y0* – the initial condition vector  $y(t_0)$ .
- *sunctx* – the [SUNContext](#) object (see §4.2)

**Return value:**

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

void **ERKStepFree**(void \*\*arkode\_mem)

This function frees the problem memory *arkode\_mem* created by [ERKStepCreate\(\)](#).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.

**Return value:** None

Deprecated since version 6.1.0: Use [ARKodeFree\(\)](#) instead.

### 5.8.1.2 ERKStep tolerance specification functions

int **ERKStepSStolerances**(void \*arkode\_mem, [sunrealtype](#) reltol, [sunrealtype](#) abstol)

This function specifies scalar relative and absolute tolerances.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – scalar absolute tolerance.

**Return value:**

- *ARK\_SUCCESS* if successful



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

Deprecated since version 6.1.0: Use [ARKodeSStolerances\(\)](#) instead.

int **ERKStepSVtolerances**(void \*arkode\_mem, *sunrealtype* reltol, *N\_Vector* abstol)

This function specifies a scalar relative tolerance and a vector absolute tolerance (a potentially different absolute tolerance for each vector component).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – vector containing the absolute tolerances for each solution component.

**Return value:**

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

Deprecated since version 6.1.0: Use [ARKodeSVtolerances\(\)](#) instead.

int **ERKStepWftolerances**(void \*arkode\_mem, *ARKEwtFn* efun)

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

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *efun* – the name of the function (of type [ARKEwtFn\(\)](#)) that implements the error weight vector computation.

**Return value:**

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

Deprecated since version 6.1.0: Use [ARKodeWftolerances\(\)](#) instead.

### 5.8.1.3 Rootfinding initialization function

int **ERKStepRootInit**(void \*arkode\_mem, int nrtfn, *ARKRootFn* g)

Initializes a rootfinding problem to be solved during the integration of the ODE system. It must be called after [ERKStepCreate\(\)](#), and before [ERKStepEvolve\(\)](#).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *nrtfn* – number of functions  $g_i$ , an integer  $\geq 0$ .
- *g* – name of user-supplied function, of type [ARKRootFn\(\)](#), defining the functions  $g_i$  whose roots are sought.

**Return value:**

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

**Notes:**

To disable the rootfinding feature after it has already been initialized, or to free memory associated with ERKStep's rootfinding module, call `ERKStepRootInit` with `nrtfn = 0`.

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

Deprecated since version 6.1.0: Use `ARKodeRootInit()` instead.

**5.8.1.4 ERKStep solver function**

int **ERKStepEvolve**(void \*arkode\_mem, *sunrealtype* tout, *N\_Vector* yout, *sunrealtype* \*tret, int itask)

Integrates the ODE over an interval in  $t$ .

**Arguments:**

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

The `ARK_NORMAL` option causes the solver to take internal steps until it has just overtaken a user-specified output time, `tout`, in the direction of integration, i.e.  $t_{n-1} < tout \leq t_n$  for forward integration, or  $t_n \leq tout < t_{n-1}$  for backward integration. It will then compute an approximation to the solution  $y(tout)$  by interpolation (using one of the dense output routines described in §2.2).

The `ARK_ONE_STEP` option tells the solver to only take a single internal step,  $y_{n-1} \rightarrow y_n$ , and return the solution at that point,  $y_n$ , in the vector `yout`.

**Return value:**

- `ARK_SUCCESS` if successful.
- `ARK_ROOT_RETURN` if `ERKStepEvolve()` succeeded, and found one or more roots. If the number of root functions, `nrtfn`, is greater than 1, call `ERKStepGetRootInfo()` to see which  $g_i$  were found to have a root at (`*tret`).
- `ARK_TSTOP_RETURN` if `ERKStepEvolve()` succeeded and returned at `tstop`.
- `ARK_MEM_NULL` if the `arkode_mem` argument was NULL.
- `ARK_NO_MALLOC` if `arkode_mem` was not allocated.
- `ARK_ILL_INPUT` if one of the inputs to `ERKStepEvolve()` is illegal, or some other input to the solver was either illegal or missing. Details will be provided in the error message. Typical causes of this failure:
  - (a) A component of the error weight vector became zero during internal time-stepping.
  - (b) A root of one of the root functions was found both at a point  $t$  and also very near  $t$ .

(c) The initial condition violates the inequality constraints.

- `ARK_TOO_MUCH_WORK` if the solver took `mxstep` internal steps but could not reach `tout`. The default value for `mxstep` is `MXSTEP_DEFAULT = 500`.
- `ARK_TOO_MUCH_ACC` if the solver could not satisfy the accuracy demanded by the user for some internal step.
- `ARK_ERR_FAILURE` if error test failures occurred either too many times (`ark_maxnef`) during one internal time step or occurred with  $|h| = h_{min}$ .
- `ARK_VECTOROP_ERR` a vector operation error occurred.

**Notes:**

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

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

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

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

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

Deprecated since version 6.1.0: Use `ARKodeEvolve()` instead.

### 5.8.1.5 Optional input functions

#### Optional inputs for ERKStep

`int ERKStepSetDefaults(void *arkode_mem)`

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

**Arguments:**

- `arkode_mem` – pointer to the ERKStep memory block.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

Does not change problem-defining function pointer `f` or the `user_data` pointer.

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

Deprecated since version 6.1.0: Use `ARKodeSetDefaults()` instead.

int **ERKStepSetInterpolantType**(void \*arkode\_mem, int itype)

Deprecated since version 6.1.0: This function is now a wrapper to [ARKodeSetInterpolantType\(\)](#), see the documentation for that function instead.

int **ERKStepSetInterpolantDegree**(void \*arkode\_mem, int degree)

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

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *degree* – requested polynomial degree.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory or interpolation module are NULL
- *ARK\_INTERP\_FAIL* if this is called after [ERKStepEvolve\(\)](#)
- *ARK\_ILL\_INPUT* if an argument had an illegal value or the interpolation module has already been initialized

**Notes:**

Allowed values are between 0 and 5.

This routine should be called *after* [ERKStepCreate\(\)](#) and *before* [ERKStepEvolve\(\)](#). After the first call to [ERKStepEvolve\(\)](#) the interpolation degree may not be changed without first calling [ERKStepReInit\(\)](#).

If a user calls both this routine and [ERKStepSetInterpolantType\(\)](#), then [ERKStepSetInterpolantType\(\)](#) must be called first.

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

Changed in version 5.5.1: When  $q = 1$ , a linear interpolant is the default to ensure values obtained by the integrator are returned at the ends of the time interval.

Deprecated since version 6.1.0: Use [ARKodeSetInterpolantDegree\(\)](#) instead.

int **ERKStepSetDenseOrder**(void \*arkode\_mem, int dord)

Deprecated since version 5.2.0: Use [ARKodeSetInterpolantDegree\(\)](#) instead.

int **ERKStepSetDiagnostics**(void \*arkode\_mem, FILE \*diagfp)

Specifies the file pointer for a diagnostics file where all ERKStep step adaptivity and solver information is written.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *diagfp* – pointer to the diagnostics output file.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had 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.

Deprecated since version 5.2.0: Use `SUNLogger_SetInfoFilename()` instead.

int **ERKStepSetFixedStep**(void \*arkode\_mem, *sunrealtype* hfixed)

Disabled time step adaptivity within ERKStep, and specifies the fixed time step size to use for the following internal step(s).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *hfixed* – value of the fixed step size to use.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

Pass 0.0 to return ERKStep to the default (adaptive-step) mode.

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

When using `ERKStepSetFixedStep()`, any values provided to the functions `ERKStepSetInitStep()`, `ERKStepSetAdaptivityFn()`, `ERKStepSetMaxErrTestFails()`, `ERKStepSetAdaptivityMethod()`, `ERKStepSetCFLFraction()`, `ERKStepSetErrorBias()`, `ERKStepSetFixedStepBounds()`, `ERKStepSetMaxEFailGrowth()`, `ERKStepSetMaxFirstGrowth()`, `ERKStepSetMaxGrowth()`, `ERKStepSetMinReduction()`, `ERKStepSetSafetyFactor()`, `ERKStepSetSmallNumEFails()`, `ERKStepSetStabilityFn()`, and `ERKStepSetAdaptController()` will be ignored, since temporal adaptivity is disabled.

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

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

Deprecated since version 6.1.0: Use `ARKodeSetFixedStep()` instead.

int **ERKStepSetInitStep**(void \*arkode\_mem, *sunrealtype* hin)

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

**Arguments:**

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

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Pass 0.0 to use the default value.

By default, ERKStep estimates the initial step size to be  $h = \sqrt{\frac{2}{\|\ddot{y}\|}}$ , where  $\ddot{y}$  is an estimate of the second derivative of the solution at  $t_0$ .

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

Deprecated since version 6.1.0: Use [ARKodeSetInitStep\(\)](#) instead.

int **ERKStepSetMaxHnilWarns**(void \*arkode\_mem, int mxhnil)

Specifies the maximum number of messages issued by the solver to warn that  $t + h = t$  on the next internal step, before ERKStep will instead return with an error.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *mxhnil* – maximum allowed number of warning messages ( $> 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had 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.

Deprecated since version 6.1.0: Use [ARKodeSetMaxHnilWarns\(\)](#) instead.

int **ERKStepSetMaxNumSteps**(void \*arkode\_mem, long int mxsteps)

Specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time, before ERKStep will return with an error.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *mxsteps* – maximum allowed number of internal steps.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Passing *mxsteps* = 0 results in ERKStep using the default value (500).

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

Deprecated since version 6.1.0: Use [ARKodeSetMaxNumSteps\(\)](#) instead.

int **ERKStepSetMaxStep**(void \*arkode\_mem, *sunrealtype* hmax)

Specifies the upper bound on the magnitude of the time step size.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *hmax* – maximum absolute value of the time step size ( $\geq 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Pass  $hmax \leq 0.0$  to set the default value of  $\infty$ .

Deprecated since version 6.1.0: Use [ARKodeSetMaxStep\(\)](#) instead.

int **ERKStepSetMinStep**(void \*arkode\_mem, *sunrealtype* hmin)

Specifies the lower bound on the magnitude of the time step size.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *hmin* – minimum absolute value of the time step size ( $\geq 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

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

Deprecated since version 6.1.0: Use [ARKodeSetMinStep\(\)](#) instead.

int **ERKStepSetStopTime**(void \*arkode\_mem, *sunrealtype* tstop)

Specifies the value of the independent variable  $t$  past which the solution is not to proceed.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *tstop* – stopping time for the integrator.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default is that no stop time is imposed.

Once the integrator returns at a stop time, any future testing for *tstop* is disabled (and can be re-enabled only through a new call to [ERKStepSetStopTime\(\)](#)).

A stop time not reached before a call to [ERKStepReInit\(\)](#) or [ERKStepReset\(\)](#) will remain active but can be disabled by calling [ERKStepClearStopTime\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetStopTime\(\)](#) instead.

int **ERKStepSetInterpolateStopTime**(void \*arkode\_mem, *sunbooleantype* interp)

Specifies that the output solution should be interpolated when the current  $t$  equals the specified  $t_{\text{stop}}$  (instead of merely copying the internal solution  $y_n$ ).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *interp* – flag indicating to use interpolation (1) or copy (0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetInterpolateStopTime\(\)](#) instead.

int **ERKStepClearStopTime**(void \*arkode\_mem)

Disables the stop time set with [ERKStepSetStopTime\(\)](#).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL

**Notes:**

The stop time can be re-enabled though a new call to [ERKStepSetStopTime\(\)](#).

Added in version 5.5.1.

Deprecated since version 6.1.0: Use [ARKodeClearStopTime\(\)](#) instead.

int **ERKStepSetUserData**(void \*arkode\_mem, void \*user\_data)

Specifies the user data block *user\_data* and attaches it to the main ERKStep memory block.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *user\_data* – pointer to the user data.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had 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.

Deprecated since version 6.1.0: Use [ARKodeSetUserData\(\)](#) instead.



int **ERKStepSetMaxErrTestFails**(void \*arkode\_mem, int maxnef)

Specifies the maximum number of error test failures permitted in attempting one step, before returning with an error.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *maxnef* – maximum allowed number of error test failures ( $> 0$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default value is 7; set *maxnef*  $\leq 0$  to specify this default.

Deprecated since version 6.1.0: Use [ARKodeSetMaxErrTestFails\(\)](#) instead.

int **ERKStepSetConstraints**(void \*arkode\_mem, *N\_Vector* constraints)

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

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *constraints* – vector of constraint flags. Each component specifies the type of solution constraint:

$$\text{constraints}[i] = \begin{cases} 0.0 & \Rightarrow \text{no constraint is imposed on } y_i, \\ 1.0 & \Rightarrow y_i \geq 0, \\ -1.0 & \Rightarrow y_i \leq 0, \\ 2.0 & \Rightarrow y_i > 0, \\ -2.0 & \Rightarrow y_i < 0. \end{cases}$$

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if the constraints vector contains illegal values

**Notes:**

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

After a call to [ERKStepResize\(\)](#) inequality constraint checking will be disabled and a call to [ERKStepSetConstraints\(\)](#) is required to re-enable constraint checking.

Since constraint-handling is performed through cutting time steps that would violate the constraints, it is possible that this feature will cause some problems to fail due to an inability to enforce constraints even at the minimum time step size. Additionally, the features [ERKStepSetConstraints\(\)](#) and [ERKStepSetFixedStep\(\)](#) are incompatible, and should not be used simultaneously.

Deprecated since version 6.1.0: Use [ARKodeSetConstraints\(\)](#) instead.

int **ERKStepSetMaxNumConstrFails**(void \*arkode\_mem, int maxfails)

Specifies the maximum number of constraint failures in a step before ERKStep will return with an error.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *maxfails* – maximum allowed number of constrain failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL

**Notes:**

Passing *maxfails*  $\leq 0$  results in ERKStep using the default value (10).

Deprecated since version 6.1.0: Use [ARKodeSetMaxNumConstrFails\(\)](#) instead.

### Optional inputs for IVP method selection

Table 5.3: Optional inputs for IVP method selection

Optional input	Function name	Default
Set integrator method order	<a href="#">ERKStepSetOrder()</a>	4
Set explicit RK table	<a href="#">ERKStepSetTable()</a>	internal
Set explicit RK table via its number	<a href="#">ERKStepSetTableNum()</a>	internal
Set explicit RK table via its name	<a href="#">ERKStepSetTableName()</a>	internal

int **ERKStepSetOrder**(void \*arkode\_mem, int ord)

Specifies the order of accuracy for the ERK integration method.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *ord* – requested order of accuracy.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The allowed values are  $2 \leq ord \leq 8$ . Any illegal input will result in the default value of 4.

Since *ord* affects the memory requirements for the internal ERKStep memory block, it cannot be changed after the first call to [ERKStepEvolve\(\)](#), unless [ERKStepReInit\(\)](#) is called.

Deprecated since version 6.1.0: Use [ARKodeSetOrder\(\)](#) instead.

int **ERKStepSetTable**(void \*arkode\_mem, [ARKodeButcherTable](#) B)

Specifies a customized Butcher table for the ERK method.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *B* – the Butcher table for the explicit RK method.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

For a description of the [ARKodeButcherTable](#) type and related functions for creating Butcher tables, see §6.

No error checking is performed to ensure that either the method order  $p$  or the embedding order  $q$  specified in the Butcher table structure correctly describe the coefficients in the Butcher table.

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

If the Butcher table does not contain an embedding, the user *must* call [ERKStepSetFixedStep\(\)](#) to enable fixed-step mode and set the desired time step size.

**Warning:**

This should not be used with [ARKodeSetOrder\(\)](#).

int **ERKStepSetTableNum**(void \*arkode\_mem, [ARKODE\\_ERKTableID](#) etable)

Indicates to use a specific built-in Butcher table for the ERK method.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *etable* – index of the Butcher table.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

*etable* should match an existing explicit method from §19.1. Error-checking is performed to ensure that the table exists, and is not implicit.

**Warning:**

This should not be used with [ARKodeSetOrder\(\)](#).

int **ERKStepSetTableName**(void \*arkode\_mem, const char \*etable)

Indicates to use a specific built-in Butcher table for the ERK method.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *etable* – name of the Butcher table.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

*etable* should match an existing explicit method from §19.1. Error-checking is performed to ensure that the table exists, and is not implicit. This function is case sensitive.

**Warning:**

This should not be used with [ARKodeSetOrder\(\)](#).

**Note**

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.table\_name”.

**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 §2.11.

int **ERKStepSetAdaptController**(void \*arkode\_mem, [SUNAdaptController C](#))

Sets a user-supplied time-step controller object.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *C* – user-supplied time adaptivity controller. If NULL then the I controller will be created (see §13.2).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_MEM\_FAIL* if *C* was NULL and the I controller could not be allocated.

Added in version 5.7.0.

Deprecated since version 6.1.0: Use [ARKodeSetAdaptController\(\)](#) instead.

Changed in version 6.3.0: The default controller was changed from PI to I. Additionally, in prior versions, passing NULL to this function would attach the PID controller.

int **ERKStepSetAdaptivityFn**(void \*arkode\_mem, [ARKAdaptFn hfun](#), void \*h\_data)

Sets a user-supplied time-step adaptivity function.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *hfun* – name of user-supplied adaptivity function.
- *h\_data* – pointer to user data passed to *hfun* every time it is called.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This function should focus on accuracy-based time step estimation; for stability based time steps the function [ERKStepSetStabilityFn\(\)](#) should be used instead.

Deprecated since version 5.7.0: Use the SUNAdaptController infrastructure instead (see §13.1).

int **ERKStepSetAdaptivityMethod**(void \*arkode\_mem, int imethod, int ndefault, int pq, [sunrealtype](#) \*adapt\_params)

Specifies the method (and associated parameters) used for time step adaptivity.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *imethod* – accuracy-based adaptivity method choice ( $0 \leq imethod \leq 5$ ): 0 is PID, 1 is PI, 2 is I, 3 is explicit Gustafsson, 4 is implicit Gustafsson, and 5 is the ImEx Gustafsson.
- *ndefault* – 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), the method order of accuracy *q* (1), or the minimum of the two (any input not equal to 0 or 1) within the adaptivity algorithm. *p* is the default.
- *adapt\_params*[0] –  $k_1$  parameter within accuracy-based adaptivity algorithms.
- *adapt\_params*[1] –  $k_2$  parameter within accuracy-based adaptivity algorithms.
- *adapt\_params*[2] –  $k_3$  parameter within accuracy-based adaptivity algorithms.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

If custom parameters are supplied, they will be checked for validity against published stability intervals. If other parameter values are desired, it is recommended to instead provide a custom function through a call to [ERKStepSetAdaptivityFn\(\)](#).

Changed in version 5.7.0: Prior to version 5.7.0, any nonzero value for *pq* would result in use of the embedding order of accuracy.

Deprecated since version 5.7.0: Use the SUNAdaptController infrastructure instead (see §13.1).

int **ERKStepSetAdaptivityAdjustment**(void \*arkode\_mem, int adjust)

Called by a user to adjust the method order supplied to the temporal adaptivity controller. For example, if the user expects order reduction due to problem stiffness, they may request that the controller assume a reduced order of accuracy for the method by specifying a value *adjust* < 0.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *adjust* – adjustment factor (default is 0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This should be called prior to calling [ERKStepEvolve\(\)](#), and can only be reset following a call to [ERKStepReInit\(\)](#).

Added in version 5.7.0.

Deprecated since version 6.1.0: Use [ARKodeSetAdaptivityAdjustment\(\)](#) instead.

Changed in version 6.3.0: The default value was changed from -1 to 0

int **ERKStepSetCFLFraction**(void \*arkode\_mem, *sunrealtype* cfl\_frac)

Specifies the fraction of the estimated explicitly stable step to use.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *cfl\_frac* – maximum allowed fraction of explicitly stable step (default is 0.5).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetCFLFraction\(\)](#) instead.

int **ERKStepSetErrorBias**(void \*arkode\_mem, *sunrealtype* bias)

Specifies the bias to be applied to the error estimates within accuracy-based adaptivity strategies.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *bias* – bias applied to error in accuracy-based time step estimation (default is 1.0).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value below 1.0 will imply a reset to the default value.

If both this and one of [ERKStepSetAdaptivityMethod\(\)](#) or [ERKStepSetAdaptController\(\)](#) will be called, then this routine must be called *second*.

Deprecated since version 5.7.0: Use the SUNAdaptController infrastructure instead (see §13.1).

Changed in version 6.3.0: The default value was changed from 1.5 to 1.0

int **ERKStepSetFixedStepBounds**(void \*arkode\_mem, *sunrealtype* lb, *sunrealtype* ub)

Specifies the step growth interval in which the step size will remain unchanged.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *lb* – lower bound on window to leave step size fixed (default is 1.0).
- *ub* – upper bound on window to leave step size fixed (default is 1.0).

**Return value:**

- *ARK\_SUCCESS* if successful

- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

Any interval *not* containing 1.0 will imply a reset to the default values.

Deprecated since version 6.1.0: Use [`ARKodeSetFixedStepBounds\(\)`](#) instead.

Changed in version 6.3.0: The default upper bound was changed from 1.5 to 1.0

int **ERKStepSetMaxEFailGrowth**(void \*arkode\_mem, *sunrealtype* etamxf)

Specifies the maximum step size growth factor upon multiple successive accuracy-based error failures in the solver.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *etamxf* – time step reduction factor on multiple error fails (default is 0.3).

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

Any value outside the interval (0, 1] will imply a reset to the default value.

Deprecated since version 6.1.0: Use [`ARKodeSetMaxEFailGrowth\(\)`](#) instead.

int **ERKStepSetMaxFirstGrowth**(void \*arkode\_mem, *sunrealtype* etamx1)

Specifies the maximum allowed growth factor in step size following the very first integration step.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *etamx1* – maximum allowed growth factor after the first time step (default is 10000.0).

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory is NULL
- `ARK_ILL_INPUT` if an argument had an illegal value

**Notes:**

Any value  $\leq 1.0$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [`ARKodeSetMaxFirstGrowth\(\)`](#) instead.

int **ERKStepSetMaxGrowth**(void \*arkode\_mem, *sunrealtype* mx\_growth)

Specifies the maximum allowed growth factor in step size between consecutive steps in the integration process.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *mx\_growth* – maximum allowed growth factor between consecutive time steps (default is 20.0).

**Return value:**

- `ARK_SUCCESS` if successful

- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value  $\leq 1.0$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetMaxGrowth\(\)](#) instead.

int **ERKStepSetMinReduction**(void \*arkode\_mem, *sunrealtype* eta\_min)

Specifies the minimum allowed reduction factor in step size between step attempts, resulting from a temporal error failure in the integration process.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *eta\_min* – minimum allowed reduction factor time step after an error test failure (default is 0.1).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any value  $\geq 1.0$  or  $\leq 0.0$  will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetMinReduction\(\)](#) instead.

int **ERKStepSetSafetyFactor**(void \*arkode\_mem, *sunrealtype* safety)

Specifies the safety factor to be applied to the accuracy-based estimated step.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *safety* – safety factor applied to accuracy-based time step (default is 0.9).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetSafetyFactor\(\)](#) instead.

Changed in version 6.3.0: The default default was changed from 0.96 to 0.9. The maximum value is now exactly 1.0 rather than strictly less than 1.0.

int **ERKStepSetSmallNumEFails**(void \*arkode\_mem, int small\_nef)

Specifies the threshold for “multiple” successive error failures before the *etamxf* parameter from [ERKStepSetMaxEFailGrowth\(\)](#) is applied.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *small\_nef* – bound to determine “multiple” for *etamxf* (default is 2).

**Return value:**



- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetSmallNumEFails\(\)](#) instead.

int **ERKStepSetStabilityFn**(void \*arkode\_mem, *ARKExpStabFn* EStab, void \*estab\_data)

Sets the problem-dependent function to estimate a stable time step size for the explicit portion of the ODE system.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *EStab* – name of user-supplied stability function.
- *estab\_data* – pointer to user data passed to *EStab* every time it is called.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

This function should return an estimate of the absolute value of the maximum stable time step for the the ODE system. It is not required, since accuracy-based adaptivity may be sufficient for retaining stability, but this can be quite useful for problems where the right-hand side function  $f(t, y)$  contains stiff terms.

Deprecated since version 6.1.0: Use [ARKodeSetStabilityFn\(\)](#) instead.

**Rootfinding optional input functions**

int **ERKStepSetRootDirection**(void \*arkode\_mem, int \*rootdir)

Specifies the direction of zero-crossings to be located and returned.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *rootdir* – state array of length *nrtfn*, the number of root functions  $g_i$  (the value of *nrtfn* was supplied in the call to [ERKStepRootInit\(\)](#)). If *rootdir*[*i*] == 0 then crossing in either direction for  $g_i$  should be reported. A value of +1 or -1 indicates that the solver should report only zero-crossings where  $g_i$  is increasing or decreasing, respectively.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL
- *ARK\_ILL\_INPUT* if an argument had an illegal value

**Notes:**

The default behavior is to monitor for both zero-crossing directions.

Deprecated since version 6.1.0: Use [ARKodeSetRootDirection\(\)](#) instead.

int **ERKStepSetNoInactiveRootWarn**(void \*arkode\_mem)

Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL

**Notes:**

ERKStep will not report the initial conditions as a possible zero-crossing (assuming that one or more components  $g_i$  are zero at the initial time). However, if it appears that some  $g_i$  is identically zero at the initial time (i.e.,  $g_i$  is zero at the initial time *and* after the first step), ERKStep will issue a warning which can be disabled with this optional input function.

Deprecated since version 6.1.0: Use [ARKodeSetNoInactiveRootWarn\(\)](#) instead.

### 5.8.1.6 Interpolated output function

int **ERKStepGetDky**(void \*arkode\_mem, [sunrealtype](#) t, int k, [N\\_Vector](#) dky)

Computes the  $k$ -th derivative of the function  $y$  at the time  $t$ , i.e.,  $y^{(k)}(t)$ , 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. This routine uses an interpolating polynomial of degree  $\min(\text{degree}, 5)$ , where *degree* is the argument provided to [ERKStepSetInterpolantDegree\(\)](#). The user may request  $k$  in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$  where  $kmax$  depends on the choice of interpolation module. For Hermite interpolants  $kmax = 5$  and for Lagrange interpolants  $kmax = 3$ .

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- $t$  – the value of the independent variable at which the derivative is to be evaluated.
- $k$  – the derivative order requested.
- *dky* – output vector (must be allocated by the user).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_BAD\_K* if  $k$  is not in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$ .
- *ARK\_BAD\_T* if  $t$  is not in the interval  $[t_n - h_n, t_n]$
- *ARK\_BAD\_DKY* if the *dky* vector was NULL
- *ARK\_MEM\_NULL* if the ERKStep memory is NULL

**Notes:**

It is only legal to call this function after a successful return from [ERKStepEvolve\(\)](#).

A user may access the values  $t_n$  and  $h_n$  via the functions [ERKStepGetCurrentTime\(\)](#) and [ERKStepGetLastStep\(\)](#), respectively.

Deprecated since version 6.1.0: Use [ARKodeGetDky\(\)](#) instead.

### 5.8.1.7 Optional output functions

#### Main solver optional output functions

int **ERKStepGetWorkspace**(void \*arkode\_mem, long int \*lenrw, long int \*leniw)

Returns the ERKStep real and integer workspace sizes.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *lenrw* – the number of *sunrealtype* values in the ERKStep workspace.
- *leniw* – the number of integer values in the ERKStep workspace.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetWorkspace\(\)](#) instead.

int **ERKStepGetNumSteps**(void \*arkode\_mem, long int \*nsteps)

Returns the cumulative number of internal steps taken by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *nsteps* – number of steps taken in the solver.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumSteps\(\)](#) instead.

int **ERKStepGetActualInitStep**(void \*arkode\_mem, *sunrealtype* \*hinused)

Returns the value of the integration step size used on the first step.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *hinused* – actual value of initial step size.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

**Notes:**

Even if the value of the initial integration step was specified by the user through a call to [ERKStepSetInitStep\(\)](#), this value may have been changed by ERKStep to ensure that the step size fell within the prescribed bounds ( $h_{min} \leq h_0 \leq h_{max}$ ), or to satisfy the local error test condition.

Deprecated since version 6.1.0: Use [ARKodeGetActualInitStep\(\)](#) instead.

int **ERKStepGetLastStep**(void \*arkode\_mem, *sunrealtype* \*hlast)

Returns the integration step size taken on the last successful internal step.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *hlast* – step size taken on the last internal step.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetLastStep\(\)](#) instead.

int **ERKStepGetCurrentStep**(void \*arkode\_mem, *sunrealtype* \*hcur)

Returns the integration step size to be attempted on the next internal step.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *hcur* – step size to be attempted on the next internal step.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentStep\(\)](#) instead.

int **ERKStepGetCurrentTime**(void \*arkode\_mem, *sunrealtype* \*tcur)

Returns the current internal time reached by the solver.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *tcur* – current internal time reached.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentTime\(\)](#) instead.

int **ERKStepGetTolScaleFactor**(void \*arkode\_mem, *sunrealtype* \*tolsfac)

Returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *tolsfac* – suggested scaling factor for user-supplied tolerances.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetTolScaleFactor\(\)](#) instead.

int **ERKStepGetErrWeights**(void \*arkode\_mem, *N\_Vector* eweight)

Returns the current error weight vector.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *eweight* – solution error weights at the current time.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

**Notes:**

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

Deprecated since version 6.1.0: Use [ARKodeGetErrWeights\(\)](#) instead.

```
int ERKStepGetStepStats(void *arkode_mem, long int *nsteps, sunrealtype *hinused, sunrealtype *hlast,
                        sunrealtype *hcur, sunrealtype *tcur)
```

Returns many of the most useful optional outputs in a single call.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *nsteps* – number of steps taken in the solver.
- *hinused* – actual value of initial step size.
- *hlast* – step size taken on the last internal step.
- *hcur* – step size to be attempted on the next internal step.
- *tcur* – current internal time reached.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetStepStats\(\)](#) instead.

```
int ERKStepPrintAllStats(void *arkode_mem, FILE *outfile, SUNOutputFormat fmt)
```

Outputs all of the integrator and other statistics.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *outfile* – pointer to output file.
- *fmt* – the output format:
  - *SUN\_OUTPUTFORMAT\_TABLE* – prints a table of values
  - *SUN\_OUTPUTFORMAT\_CSV* – prints a comma-separated list of key and value pairs e.g., key1, value1, key2, value2, ...

**Return value:**

- *ARK\_SUCCESS* – if the output was successfully.
- *CV\_MEM\_NULL* – if the ERKStep memory was NULL.
- *CV\_ILL\_INPUT* – if an invalid formatting option was provided.

**Note**

The Python module `tools/suntools` provides utilities to read and output the data from a SUNDIALS CSV output file using the key and value pair format.

Added in version 5.2.0.

Deprecated since version 6.1.0: Use [`ARKodePrintAllStats\(\)`](#) instead.

char \***ERKStepGetReturnFlagName**(long int flag)

Returns the name of the ERKStep constant corresponding to *flag*. See [\*ARKODE Constants\*](#).

**Arguments:**

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

**Return value:**

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

**Warning**

The user is responsible for freeing the returned string.

Deprecated since version 6.1.0: Use [`ARKodeGetReturnFlagName\(\)`](#) instead.

int **ERKStepGetNumExpSteps**(void \*arkode\_mem, long int \*expsteps)

Returns the cumulative number of stability-limited steps taken by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [`ARKodeGetNumExpSteps\(\)`](#) instead.

int **ERKStepGetNumAccSteps**(void \*arkode\_mem, long int \*accsteps)

Returns the cumulative number of accuracy-limited steps taken by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *accsteps* – number of accuracy-limited steps taken in the solver.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [`ARKodeGetNumAccSteps\(\)`](#) instead.

int **ERKStepGetNumStepAttempts**(void \*arkode\_mem, long int \*step\_attempts)

Returns the cumulative number of steps attempted by the solver (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *step\_attempts* – number of steps attempted by solver.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumStepAttempts\(\)](#) instead.

int **ERKStepGetNumRhsEvals**(void \*arkode\_mem, long int \*nf\_evals)

Returns the number of calls to the user's right-hand side function,  $f$  (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *nf\_evals* – number of calls to the user's  $f(t, y)$  function.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.2.0: Use [ARKodeGetNumRhsEvals\(\)](#) instead.

int **ERKStepGetNumErrTestFails**(void \*arkode\_mem, long int \*netfails)

Returns the number of local error test failures that have occurred (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *netfails* – number of error test failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumErrTestFails\(\)](#) instead.

int **ERKStepGetCurrentButcherTable**(void \*arkode\_mem, [ARKodeButcherTable](#) \*B)

Returns the Butcher table currently in use by the solver.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *B* – pointer to the Butcher table structure.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

**Notes:**

The [ARKodeButcherTable](#) data structure is defined as a pointer to the following C structure:

```
typedef struct ARKodeButcherTableMem {  
  
    int q;           /* method order of accuracy */  
    int p;           /* embedding order of accuracy */  
    int stages;      /* number of stages */  
    sunrealtype **A; /* Butcher table coefficients */  
    sunrealtype *c;  /* canopy node coefficients */  
    sunrealtype *b;  /* root node coefficients */  
    sunrealtype *d;  /* embedding coefficients */  
  
} *ARKodeButcherTable;
```

For more details see §6.

int **ERKStepGetEstLocalErrors**(void \*arkode\_mem, *N\_Vector* ele)

Returns the vector of estimated local truncation errors for the current step.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *ele* – vector of estimated local truncation errors.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

**Notes:**

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

The values returned in *ele* are valid only after a successful call to [ERKStepEvolve\(\)](#) (i.e., it returned a non-negative value).

The *ele* vector, together with the *eweight* vector from [ERKStepGetErrWeights\(\)](#), can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the WRMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as *eweight[i]\*ele[i]*.

Deprecated since version 6.1.0: Use [ARKodeGetEstLocalErrors\(\)](#) instead.

int **ERKStepGetTimestepperStats**(void \*arkode\_mem, long int \*expsteps, long int \*accsteps, long int \*step\_attempts, long int \*nf\_evals, long int \*netfails)

Returns many of the most useful time-stepper statistics in a single call.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.
- *accsteps* – number of accuracy-limited steps taken in the solver.
- *step\_attempts* – number of steps attempted by the solver.
- *nf\_evals* – number of calls to the user's  $f(t, y)$  function.
- *netfails* – number of error test failures.

**Return value:**



- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

int **ERKStepGetNumConstrFails**(void \*arkode\_mem, long int \*nconstrfails)

Returns the cumulative number of constraint test failures (so far).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *nconstrfails* – number of constraint test failures.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumConstrFails\(\)](#) instead.

int **ERKStepGetUserData**(void \*arkode\_mem, void \*\*user\_data)

Returns the user data pointer previously set with [ERKStepSetUserData\(\)](#).

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *user\_data* – memory reference to a user data pointer

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Added in version 5.3.0.

Deprecated since version 6.1.0: Use [ARKodeGetUserData\(\)](#) instead.

## Rootfinding optional output functions

int **ERKStepGetRootInfo**(void \*arkode\_mem, int \*rootsfound)

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

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *rootsfound* – array of length *nrtfn* with the indices of the user functions  $g_i$  found to have a root (the value of *nrtfn* was supplied in the call to [ERKStepRootInit\(\)](#)). For  $i = 0 \dots nrtfn-1$ , *rootsfound*[*i*] is nonzero if  $g_i$  has a root, and 0 if not.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

**Notes:**

The user must allocate space for *rootsfound* prior to calling this function.

For the components of  $g_i$  for which a root was found, the sign of *rootsfound*[*i*] indicates the direction of zero-crossing. A value of +1 indicates that  $g_i$  is increasing, while a value of -1 indicates a decreasing  $g_i$ .

Deprecated since version 6.1.0: Use [ARKodeGetRootInfo\(\)](#) instead.

int **ERKStepGetNumGEvals**(void \*arkode\_mem, long int \*ngevals)

Returns the cumulative number of calls made to the user's root function  $g$ .

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *ngevals* – number of calls made to  $g$  so far.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumGEvals\(\)](#) instead.

## General usability functions

int **ERKStepWriteParameters**(void \*arkode\_mem, FILE \*fp)

Outputs all ERKStep solver parameters to the provided file pointer.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *fp* – pointer to use for printing the solver parameters.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

**Notes:**

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

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

Deprecated since version 6.1.0: Use [ARKodeWriteParameters\(\)](#) instead.

int **ERKStepWriteButcher**(void \*arkode\_mem, FILE \*fp)

Outputs the current Butcher table to the provided file pointer.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *fp* – pointer to use for printing the Butcher table.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL

**Notes:**

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

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

Deprecated since version 6.1.0: Use [ERKStepGetCurrentButcherTable\(\)](#) and [ARKodeButcherTable\\_Write\(\)](#) instead.

### 5.8.1.8 ERKStep re-initialization function

To reinitialize the ERKStep module for the solution of a new problem, where a prior call to `ERKStepCreate()` has been made, the user must call the function `ERKStepReInit()`. The new problem must have the same size as the previous one. This routine retains the current settings for all ERKStep module options and performs the same input checking and initializations that are done in `ERKStepCreate()`, but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration, and deletes any previously-set *tstop* value specified via a call to `ERKStepSetStopTime()`. Following a successful call to `ERKStepReInit()`, call `ERKStepEvolve()` again for the solution of the new problem.

The use of `ERKStepReInit()` requires that the number of Runge–Kutta stages, denoted by *s*, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the method order *q* is left unchanged.

One important use of the `ERKStepReInit()` function is in the treating of jump discontinuities in the RHS function. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to this routine. To stop when the location of the discontinuity is known, simply make that location a value of *tout*. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS function *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS function (communicated through *user\_data*) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **ERKStepReInit**(void \*arkode\_mem, *ARKRhsFn* f, *sunrealtype* t0, *N\_Vector* y0)

Provides required problem specifications and re-initializes the ERKStep time-stepper module.

#### Arguments:

- *arkode\_mem* – pointer to the ERKStep memory block.
- *f* – the name of the C function (of type `ARKRhsFn()`) defining the right-hand side function in  $\dot{y} = f(t, y)$ .
- *t0* – the initial value of *t*.
- *y0* – the initial condition vector  $y(t_0)$ .

#### Return value:

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the ERKStep memory was NULL
- `ARK_MEM_FAIL` if a memory allocation failed
- `ARK_ILL_INPUT` if an argument had an illegal value.

#### Notes:

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, `ERKStepReInit()` also sends an error message to the error handler function.

### 5.8.1.9 ERKStep reset function

int **ERKStepReset**(void \*arkode\_mem, *sunrealtype* tR, *N\_Vector* yR)

Resets the current ERKStep time-stepper module state to the provided independent variable value and dependent variable vector.

#### Arguments:

- *arkode\_mem* – pointer to the ERKStep memory block.
- *tR* – the value of the independent variable  $t$ .
- *yR* – the value of the dependent variable vector  $y(t_R)$ .

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL
- *ARK\_MEM\_FAIL* if a memory allocation failed
- *ARK\_ILL\_INPUT* if an argument had an illegal value.

**Notes:**

By default the next call to [ERKStepEvolve\(\)](#) will use the step size computed by ERKStep prior to calling [ERKStepReset\(\)](#). To set a different step size or have ERKStep estimate a new step size use [ERKStepSetInitStep\(\)](#).

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, [ERKStepReset\(\)](#) also sends an error message to the error handler function.

Deprecated since version 6.1.0: Use [ARKodeReset\(\)](#) instead.

#### 5.8.1.10 ERKStep system resize function

int **ERKStepResize**(void \*arkode\_mem, *N\_Vector* yR, *sunrealtype* hscale, *sunrealtype* tR, *ARKVecResizeFn* resize, void \*resize\_data)

Re-sizes ERKStep with a different state vector but with comparable dynamical time scale.

**Arguments:**

- *arkode\_mem* – pointer to the ERKStep memory block.
- *yR* – the newly-sized solution vector, holding the current dependent variable values  $y(t_R)$ .
- *hscale* – the desired time step scaling factor (i.e. the next step will be of size  $h*hscale$ ).
- *tR* – the current value of the independent variable  $t_R$  (this must be consistent with *yR*).
- *resize* – the user-supplied vector resize function (of type [ARKVecResizeFn\(\)](#)).
- *resize\_data* – the user-supplied data structure to be passed to *resize* when modifying internal ERKStep vectors.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the ERKStep memory was NULL
- *ARK\_NO\_MALLOC* if *arkode\_mem* was not allocated.
- *ARK\_ILL\_INPUT* if an argument had an illegal value.

**Notes:**

If an error occurred, [ERKStepResize\(\)](#) also sends an error message to the error handler function.

If inequality constraint checking is enabled a call to [ERKStepResize\(\)](#) will disable constraint checking.

A call to [ERKStepSetConstraints\(\)](#) is required to re-enable constraint checking.

Deprecated since version 6.1.0: Use [ARKodeResize\(\)](#) instead.

## 5.8.2 Relaxation Methods

This section describes ERKStep-specific user-callable functions for applying relaxation methods with ERKStep. All of these routines have been deprecated in favor of *shared ARKODE-level routines*, but this documentation will be retained for as long as these functions are present

### 5.8.2.1 Enabling or Disabling Relaxation

int **ERKStepSetRelaxFn**(void \*arkode\_mem, *ARKRelaxFn* rfn, *ARKRelaxJacFn* rjac)

Attaches the user supplied functions for evaluating the relaxation function (*rfn*) and its Jacobian (*rjac*).

Both *rfn* and *rjac* are required and an error will be returned if only one of the functions is NULL. If both *rfn* and *rjac* are NULL, relaxation is disabled.

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **rfn** – the user-defined function to compute the relaxation function  $\xi(y)$
- **rjac** – the user-defined function to compute the relaxation Jacobian  $\xi'(y)$

#### Return values

- **ARK\_SUCCESS** – the function exited successfully
- **ARK\_MEM\_NULL** – *arkode\_mem* was NULL
- **ARK\_ILL\_INPUT** – an invalid input combination was provided (see the output error message for more details)
- **ARK\_MEM\_FAIL** – a memory allocation failed

#### Warning

Applying relaxation requires using a method of at least second order with  $b_i \geq 0$ . If these conditions are not satisfied, *ERKStepEvolve()* will return with an error during initialization.

#### Note

When combined with fixed time step sizes, ERKStep will attempt each step using the specified step size. If the step is successful, relaxation will be applied, effectively modifying the step size for the current step. If the step fails or applying relaxation fails, *ERKStepEvolve()* will return with an error.

Added in version 5.6.0.

Deprecated since version 6.1.0: Use *ARKodeSetRelaxFn()* instead.

### 5.8.2.2 Optional Input Functions

This section describes optional input functions used to control applying relaxation.

int **ERKStepSetRelaxEtaFail**(void \*arkode\_mem, *sunrealtype* eta\_rf)

Sets the step size reduction factor applied after a failed relaxation application.

The default value is 0.25. Input values  $\leq 0$  or  $\geq 1$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ERKStep memory structure
- **eta\_rf** – the step size reduction factor

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxEtaFail\(\)](#) instead.

int **ERKStepSetRelaxLowerBound**(void \*arkode\_mem, *sunrealtype* lower)

Sets the smallest acceptable value for the relaxation parameter.

Values smaller than the lower bound will result in a failed relaxation application and the step will be repeated with a smaller step size (determined by [ERKStepSetRelaxEtaFail\(\)](#)).

The default value is 0.8. Input values  $\leq 0$  or  $\geq 1$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ERKStep memory structure
- **lower** – the relaxation parameter lower bound

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxLowerBound\(\)](#) instead.

int **ERKStepSetRelaxUpperBound**(void \*arkode\_mem, *sunrealtype* upper)

Sets the largest acceptable value for the relaxation parameter.

Values larger than the upper bound will result in a failed relaxation application and the step will be repeated with a smaller step size (determined by [ERKStepSetRelaxEtaFail\(\)](#)).

The default value is 1.2. Input values  $\leq 1$  will result in the default value being used.

**Parameters**

- **arkode\_mem** – the ERKStep memory structure
- **upper** – the relaxation parameter upper bound

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxUpperBound\(\)](#) instead.

int **ERKStepSetRelaxMaxFails**(void \*arkode\_mem, int max\_fails)

Sets the maximum number of times applying relaxation can fail within a step attempt before the integration is halted with an error.

The default value is 10. Input values  $\leq 0$  will result in the default value being used.

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **max\_fails** – the maximum number of failed relaxation applications allowed in a step

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxMaxFails\(\)](#) instead.

int **ERKStepSetRelaxMaxIters**(void \*arkode\_mem, int max\_iters)

Sets the maximum number of nonlinear iterations allowed when solving for the relaxation parameter.

If the maximum number of iterations is reached before meeting the solve tolerance (determined by [ERKStepSetRelaxResTol\(\)](#) and [ERKStepSetRelaxTol\(\)](#)), the step will be repeated with a smaller step size (determined by [ERKStepSetRelaxEtaFail\(\)](#)).

The default value is 10. Input values  $\leq 0$  will result in the default value being used.

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **max\_iters** – the maximum number of solver iterations allowed

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxMaxIters\(\)](#) instead.

int **ERKStepSetRelaxSolver**(void \*arkode\_mem, [ARKRelaxSolver](#) solver)

Sets the nonlinear solver method used to compute the relaxation parameter.

The default value is [ARK\\_RELAX\\_NEWTON](#)

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **solver** – the nonlinear solver to use

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

- **ARK\_ILL\_INPUT** – an invalid solver option was provided

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxSolver\(\)](#) instead.

int **ERKStepSetRelaxResTol**(void \*arkode\_mem, *sunrealtype* res\_tol)

Sets the nonlinear solver residual tolerance to use when solving (2.63).

If the residual or solution tolerance (see [ERKStepSetRelaxMaxIters\(\)](#)) is not reached within the maximum number of iterations (determined by [ERKStepSetRelaxMaxIters\(\)](#)), the step will be repeated with a smaller step size (determined by [ERKStepSetRelaxEtaFail\(\)](#)).

The default value is  $4\epsilon$  where  $\epsilon$  is floating-point precision. Input values  $\leq 0$  will result in the default value being used.

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **res\_tol** – the nonlinear solver residual tolerance to use

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxResTol\(\)](#) instead.

int **ERKStepSetRelaxTol**(void \*arkode\_mem, *sunrealtype* rel\_tol, *sunrealtype* abs\_tol)

Sets the nonlinear solver relative and absolute tolerance on changes in  $r$  when solving (2.63).

If the residual (see [ERKStepSetRelaxResTol\(\)](#)) or solution tolerance is not reached within the maximum number of iterations (determined by [ERKStepSetRelaxMaxIters\(\)](#)), the step will be repeated with a smaller step size (determined by [ERKStepSetRelaxEtaFail\(\)](#)).

The default relative and absolute tolerances are  $4\epsilon$  and  $10^{-14}$ , respectively, where  $\epsilon$  is floating-point precision. Input values  $\leq 0$  will result in the default value being used.

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **rel\_tol** – the nonlinear solver relative solution tolerance to use
- **abs\_tol** – the nonlinear solver absolute solution tolerance to use

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetRelaxTol\(\)](#) instead.



### 5.8.2.3 Optional Output Functions

This section describes optional output functions used to retrieve information about the performance of the relaxation method.

int **ERKStepGetNumRelaxFnEvals**(void \*arkode\_mem, long int \*r\_evals)

Get the number of times the user's relaxation function was evaluated.

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **r\_evals** – the number of relaxation function evaluations

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxFnEvals\(\)](#) instead.

int **ERKStepGetNumRelaxJacEvals**(void \*arkode\_mem, long int \*J\_evals)

Get the number of times the user's relaxation Jacobian was evaluated.

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **J\_evals** – the number of relaxation Jacobian evaluations

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxJacEvals\(\)](#) instead.

int **ERKStepGetNumRelaxFails**(void \*arkode\_mem, long int \*fails)

Get the total number of times applying relaxation failed.

The counter includes the sum of the number of nonlinear solver failures (see [ERKStepGetNumRelaxSolveFails\(\)](#)) and the number of failures due an unacceptable relaxation value (see [ERKStepSetRelaxLowerBound\(\)](#) and [ERKStepSetRelaxUpperBound\(\)](#)).

#### Parameters

- **arkode\_mem** – the ERKStep memory structure
- **fails** – the total number of failed relaxation attempts

#### Return values

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxFails\(\)](#) instead.

int **ERKStepGetNumRelaxBoundFails**(void \*arkode\_mem, long int \*fails)

Get the number of times the relaxation parameter was deemed unacceptable.

**Parameters**

- **arkode\_mem** – the ERKStep memory structure
- **fails** – the number of failures due to an unacceptable relaxation parameter value

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxBoundFails\(\)](#) instead.

int **ERKStepGetNumRelaxSolveFails**(void \*arkode\_mem, long int \*fails)

Get the number of times the relaxation parameter nonlinear solver failed.

**Parameters**

- **arkode\_mem** – the ERKStep memory structure
- **fails** – the number of relaxation nonlinear solver failures

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxSolveFails\(\)](#) instead.

int **ERKStepGetNumRelaxSolveIters**(void \*arkode\_mem, long int \*iters)

Get the number of relaxation parameter nonlinear solver iterations.

**Parameters**

- **arkode\_mem** – the ERKStep memory structure
- **iters** – the number of relaxation nonlinear solver iterations

**Return values**

- **ARK\_SUCCESS** – the value was successfully set
- **ARK\_MEM\_NULL** – arkode\_mem was NULL
- **ARK\_RELAX\_MEM\_NULL** – the internal relaxation memory structure was NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeGetNumRelaxSolveIters\(\)](#) instead.

## 5.9 Using the ForcingStep time-stepping module

This section is concerned with the use of the ForcingStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. Usage of ForcingStep follows that of the rest of ARKODE, and so in this section we primarily focus on those usage aspects that are specific to ForcingStep. A skeleton of a program using ForcingStep follows essentially the same structure as SplittingStep (see §5.12.1).

### 5.9.1 ForcingStep User-callable functions

This section describes the ForcingStep-specific functions that may be called by the user to setup and then solve an IVP using the ForcingStep time-stepping module.

As discussed in the main *ARKODE user-callable function introduction*, each of ARKODE's time-stepping modules clarifies the categories of user-callable functions that it supports. ForcingStep does not support any of the categories beyond the functions that apply for all time-stepping modules.

#### 5.9.1.1 ForcingStep initialization functions

void \***ForcingStepCreate**(*SUNStepper* stepper1, *SUNStepper* stepper2, *sunrealtype* t0, *N\_Vector* y0, *SUNContext* sunctx)

This function allocates and initializes memory for a problem to be solved using the ForcingStep time-stepping module in ARKODE.

##### Parameters

- **stepper1** – A *SUNStepper* to integrate partition one. At minimum, it must implement the *SUNStepper\_Evolve()*, *SUNStepper\_Reset()*, and *SUNStepper\_SetStopTime()* operations.
- **stepper2** – A *SUNStepper* to integrate partition two including the forcing from partition one. At minimum, it must implement the *SUNStepper\_Evolve()*, *SUNStepper\_Reset()*, *SUNStepper\_SetStopTime()*, and *SUNStepper\_SetForcing()* operations.
- **t0** – The initial value of  $t$ .
- **y0** – The initial condition vector  $y(t_0)$ .
- **sunctx** – The *SUNContext* object (see §4.2)

##### Returns

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

##### Example usage:

```
/* inner ARKODE objects for integrating individual partitions */
void *partition_mem[] = {NULL, NULL};

/* SUNSteppers to wrap the inner ARKStep objects */
SUNStepper steppers[] = {NULL, NULL};

/* create ARKStep objects, setting right-hand side functions and the
   initial condition */
partition_mem[0] = ARKStepCreate(fe1, fi1, t0, y0, sunctx);
```

(continues on next page)

(continued from previous page)

```
partition_mem[1] = ARKStepCreate(fe2, fi2, t0, y0, sunctx);

/* setup ARKStep */
. . .

/* create SUNStepper wrappers for the ARKStep memory blocks */
flag = ARKodeCreateSUNStepper(partition_mem[0], &stepper[0]);
flag = ARKodeCreateSUNStepper(partition_mem[1], &stepper[1]);

/* create a ForcingStep object */
arkode_mem = ForcingStepCreate(steppers[0], steppers[1], t0, y0, sunctx);
```

**Example codes:**

- `examples/arkode/C_serial/ark_analytic_partitioned.c`

Added in version 6.2.0.

**5.9.1.2 Optional output functions****int ForcingStepGetNumEvolves**(void \*arkode\_mem, int partition, long int \*evolves)Returns the number of times the *SUNStepper* for the given partition index has been evolved (so far).**Parameters**

- **arkode\_mem** – pointer to the ForcingStep memory block.
- **partition** – index of the partition (0 or 1) or a negative number to indicate the total number across both partitions.
- **evolves** – number of *SUNStepper* evolves.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the ForcingStep memory was NULL
- **ARK\_ILL\_INPUT** – if *partition* was out of bounds

Added in version 6.2.0.

**5.9.1.3 ForcingStep re-initialization function**

To reinitialize the ForcingStep module for the solution of a new problem, where a prior call to *ForcingStepCreate()* has been made, the user must call the function *ForcingStepReInit()* and re-initialize each *SUNStepper*. The new problem must have the same size as the previous one. This routine retains the current settings for all ForcingStep module options and performs the same input checking and initializations that are done in *ForcingStepCreate()*, but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration, and deletes any previously-set *tstop* value specified via a call to *ARKodeSetStopTime()*. Following a successful call to *ForcingStepReInit()*, call *ARKodeEvolve()* again for the solution of the new problem.

One important use of the *ForcingStepReInit()* function is in the treating of jump discontinuities in the RHS function. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to this routine. To stop when the location of the discontinuity

is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS function *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS function (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

Another use of `ForcingStepReInit()` is changing the partitioning of the ODE and the `SUNStepper` objects used to evolve each partition.

```
int ForcingStepReInit(void *arkode_mem, SUNStepper stepper1, SUNStepper stepper2, sunrealtype t0,
                     N_Vector y0)
```

Provides required problem specifications and re-initializes the ForcingStep time-stepper module.

#### Parameters

- **arkode\_mem** – pointer to the ForcingStep memory block.
- **stepper1** – A `SUNStepper` to integrate partition one. At minimum, it must implement the `SUNStepper_Evolve()`, `SUNStepper_Reset()`, and `SUNStepper_SetStopTime()` operations.
- **stepper2** – A `SUNStepper` to integrate partition two including the forcing from partition one. At minimum, it must implement the `SUNStepper_Evolve()`, `SUNStepper_Reset()`, `SUNStepper_SetStopTime()`, and `SUNStepper_SetForcing()` operations.
- **t0** – The initial value of  $t$ .
- **y0** – The initial condition vector  $y(t_0)$ .

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the ForcingStep memory was NULL
- **ARK\_MEM\_FAIL** – if a memory allocation failed
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Warning

This function does not perform any re-initialization of the `SUNStepper` objects. It is up to the user to do this, if necessary.

#### Note

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

Added in version 6.2.0.

## 5.10 Using the LSRKStep time-stepping module

This section is concerned with the use of the LSRKStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. Usage of LSRKStep follows that of the rest of ARKODE, and so in this section we primarily focus on those usage aspects that are specific to LSRKStep.

### 5.10.1 LSRKStep User-callable functions

This section describes the LSRKStep-specific functions that may be called by the user to setup and then solve an IVP using the LSRKStep time-stepping module. As mentioned in Section §5.3, shared ARKODE-level routines may be used for the large majority of LSRKStep configuration and use. In this section, we describe only those routines that are specific to LSRKStep.

As discussed in the main [ARKODE user-callable function introduction](#), each of ARKODE's time-stepping modules clarifies the categories of user-callable functions that it supports. LSRKStep supports the following categories:

- temporal adaptivity

LSRKStep does not have forcing function support when converted to a [SUNStepper](#) or [MRISStepInnerStepper](#). See [ARKodeCreateSUNStepper\(\)](#) and [ARKStepCreateMRISStepInnerStepper\(\)](#) for additional details.

#### 5.10.1.1 LSRKStep initialization functions

void **\*LSRKStepCreateSTS**(*ARKRhsFn* rhs, *sunrealtype* t0, *N\_Vector* y0, *SUNContext* suncctx);

This function allocates and initializes memory for a problem to be solved using STS methods from the LSRKStep time-stepping module in ARKODE.

**Arguments:**

- *rhs* – the name of the C function (of type [ARKRhsFn\(\)](#)) defining the right-hand side function.
- *t0* – the initial value of  $t$ .
- *y0* – the initial condition vector  $y(t_0)$ .
- *suncctx* – the [SUNContext](#) object (see §4.2)

**Return value:**

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

void **\*LSRKStepCreateSSP**(*ARKRhsFn* rhs, *sunrealtype* t0, *N\_Vector* y0, *SUNContext* suncctx);

This function allocates and initializes memory for a problem to be solved using SSP methods from the LSRKStep time-stepping module in ARKODE.

**Arguments:**

- *rhs* – the name of the C function (of type [ARKRhsFn\(\)](#)) defining the right-hand side function.
- *t0* – the initial value of  $t$ .
- *y0* – the initial condition vector  $y(t_0)$ .
- *suncctx* – the [SUNContext](#) object (see §4.2)

**Return value:**

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

#### 5.10.1.2 Optional input functions

```
int LSRKStepSetSTSMethod(void *arkode_mem, ARKODE_LSRKMethodType method);
```

This function selects the LSRK STS method that should be used. The list of allowable values for this input is below. *LSRKStepCreateSTS()* defaults to using *ARKODE\_LSRK\_RKC\_2*.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *method* – Type of the method.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. typo in the method type).

```
int LSRKStepSetSSPMethod(void *arkode_mem, ARKODE_LSRKMethodType method);
```

This function selects the LSRK SSP method that should be used. The list of allowable values for this input is below. *LSRKStepCreateSSP()* defaults to using *ARKODE\_LSRK\_SSP\_S\_2*.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *method* – Type of the method.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. typo in the method type).

Allowable Method Families

enum **ARKODE\_LSRKMethodType**

enumerator **ARKODE\_LSRK\_RKC\_2**

Second order Runge–Kutta–Chebyshev method

enumerator **ARKODE\_LSRK\_RKL\_2**

Second order Runge–Kutta–Legendre method

enumerator **ARKODE\_LSRK\_SSP\_S\_2**

Second order, s-stage SSP(s,2) method

enumerator **ARKODE\_LSRK\_SSP\_S\_3**

Third order, s-stage SSP(s,3) method

enumerator **ARKODE\_LSRK\_SSP\_10\_4**

Fourth order, 10-stage SSP(10,4) method

```
int LSRKStepSetSTSMethodByName(void *arkode_mem, const char *emethod);
```

This function selects the LSRK STS method by name. The list of allowable values for this input is above. *LSRKStepCreateSTS()* defaults to using *ARKODE\_LSRK\_RKC\_2*.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *emethod* – the method name.

**Return value:**

- *ARK\_SUCCESS* if successful

- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. typo in the method name).

**Note**

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.sts\_method\_name”.

int **LSRKStepSetSSPMethodByName**(void \*arkode\_mem, const char \*emethod);

This function selects the LSRK SSP method by name. The list of allowable values for this input is above. *LSRKStepCreateSSP()* defaults to using *ARKODE\_LSRK\_SSP\_S\_2*.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *emethod* – the method name.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g. typo in the method name).

**Note**

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.ssp\_method\_name”.

int **LSRKStepSetDomEigFn**(void \*arkode\_mem, *ARKDomEigFn* dom\_eig);

Specifies the user-supplied dominant eigenvalue approximation routine to be used for determining the number of stages that will be used by either the RKC or RKL methods.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *dom\_eig* – name of user-supplied dominant eigenvalue approximation function (of type *ARKDomEigFn()*).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if *arkode\_mem* was NULL.

**Note**

When using RKC or RKL methods, users must supply a *ARKDomEigFn* function or attach a dominant eigenvalue estimator with *LSRKStepSetDomEigEstimator()*.

int **LSRKStepSetDomEigEstimator**(void \*arkode\_mem, *SUNDomEigEstimator* DEE);

Specifies the dominant eigenvalue estimator (DEE) used to determine the number of stages in an RKC or RKL method. This function is an alternative to supplying a dominant eigenvalue function with *LSRKStepSetDomEigFn()*.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *DEE* – the dominant eigenvalue estimator to use.



**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if *arkode\_mem* was NULL.
- *ARK\_ILL\_INPUT* if an argument had an illegal value (e.g., DEE does not implement the required operations)
- *ARK\_DEE\_FAIL* if the call to *SUNDomEigEstimator\_SetATimes()* failed

Added in version 6.5.0.

**Note**

When using RKC or RKL methods, users must supply a *ARKDomEigFn* function or attach a dominant eigenvalue estimator with *LSRKStepSetDomEigEstimator()*. If both are provided then the estimator DEE will be used and the function ignored.

ARKODE will supply the *SUNDomEigEstimator* with an internal Jacobian-vector product approximation function. Users may supply their own Jacobian-vector product function by calling *SUNDomEigEstimator\_SetATimes()* after attaching the estimator with *LSRKStepSetDomEigEstimator()*.

int **LSRKStepSetDomEigFrequency**(void \*arkode\_mem, long int nsteps);

Specifies the number of steps after which the dominant eigenvalue information is considered out-of-date, and should be recomputed. This only applies to RKL and RKC methods.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *nsteps* – the dominant eigenvalue re-computation update frequency. A value *nsteps* = 0 indicates that the dominant eigenvalue will not change throughout the simulation.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if *arkode\_mem* was NULL.

**Note**

If LSRKStepSetDomEigFrequency routine is not called, then the default *nsteps* is set to 25 as recommended in [122]. Calling this function with *nsteps* < 0 resets the default value while *nsteps* = 0 refers to constant dominant eigenvalue.

Calling this function with *nsteps* < 0 resets the default value while *nsteps* = 0 refers to constant dominant eigenvalue.

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.dom\_eig\_frequency”.

int **LSRKStepSetMaxNumStages**(void \*arkode\_mem, int stage\_max\_limit);

Specifies the maximum number of stages allowed within each time step. This bound only applies to RKL and RKC methods.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *stage\_max\_limit* – maximum allowed number of stages ( $\geq 2$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if *arkode\_mem* was NULL.

**Note**

If *LSRKStepSetMaxNumStages()* is not called, the default *stage\_max\_limit* is set to 200. Calling this function with *stage\_max\_limit* < 2 resets the default value.

This limit should be chosen with consideration of the following proportionality:  $s^2 \sim -h\lambda$ , where  $s$  is the number of stages used,  $h$  is the current step size and  $\lambda$  is the dominant eigenvalue.

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.max\_num\_stages”.

int **LSRKStepSetDomEigSafetyFactor**(void \*arkode\_mem, *sunrealtype* dom\_eig\_safety);

Specifies a safety factor to use for the result of the dominant eigenvalue estimation function. This value is used to scale the magnitude of the dominant eigenvalue, in the hope of ensuring a sufficient number of stages for the method to be stable. This input is only used for RKC and RKL methods.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *dom\_eig\_safety* – safety factor ( $\geq 1$ ).

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if *arkode\_mem* was NULL.

**Note**

If *LSRKStepSetDomEigSafetyFactor()* is not called, then the default *dom\_eig\_safety* is set to 1.01. Calling this function with *dom\_eig\_safety* < 1 resets the default value.

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.dom\_eig\_safety\_factor”.

int **LSRKStepSetNumDomEigEstInitPreprocessIters**(void \*arkode\_mem, int num\_iters);

Specifies the number of the preprocessing iterations before the very first estimate call.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *num\_iters* – the number of iterations.

**Return value:**

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if *arkode\_mem* was NULL.

Added in version 6.5.0.

**Note**

If `LSRKStepSetNumDomEigEstInitPreprocessIters` routine is not called, then the default value of the estimator is used. Calling this function with `num_iters < 0` resets the default.

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.num\_dom\_eig\_est\_init\_preprocess\_iters”.

int **LSRKStepSetNumDomEigEstPreprocessIters**(void \*arkode\_mem, int num\_iters);

Specifies the number of the preprocessing iterations before each estimate call after the very first estimate.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *num\_iters* – the number of iterations.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if *arkode\_mem* was NULL.
- `ARK_DEE_FAIL` if the call to `SUNDomEigEstimator_SetNumPreprocessIters()` failed.

Added in version 6.5.0.

**Note**

If `LSRKStepSetNumDomEigEstPreprocessIters` routine is not called, then the default value of 0 is used. Calling this function with `num_iters < 0` resets the default.

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.num\_dom\_eig\_est\_preprocess\_iters”.

int **LSRKStepSetNumSSPStages**(void \*arkode\_mem, int num\_of\_stages);

Sets the number of stages, *s* in SSP(*s*, *p*) methods. This input is only utilized by SSPRK methods.

- `ARKODE_LSRK_SSP_S_2` – *num\_of\_stages* must be greater than or equal to 2
- `ARKODE_LSRK_SSP_S_3` – *num\_of\_stages* must be a perfect-square greater than or equal to 4
- `ARKODE_LSRK_SSP_10_4` – *num\_of\_stages* cannot be modified from 10, so this function should not be called.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *num\_of\_stages* – number of stages ( $> 1$ ) for SSP(*s*, 2) and ( $n^2 = s \geq 4$ ) for SSP(*s*, 3).

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if *arkode\_mem* was NULL.
- `ARK_ILL_INPUT` if an argument had an illegal value (e.g. SSP method is not declared)

**Note**

If `LSRKStepSetNumSSPStages()` is not called, the default `num_of_stages` is set. Calling this function with `num_of_stages <= 0` resets the default values:

- `num_of_stages = 2` for `ARKODE_LSRK_SSP_S_2`
- `num_of_stages = 4` for `ARKODE_LSRK_SSP_S_3`
- `num_of_stages = 10` for `ARKODE_LSRK_SSP_10_4`

This routine will be called by `ARKodeSetOptions()` when using the key “arkid.num\_ssp\_stages”.

Changed in version 6.7.0: The default number of stages for `ARKODE_LSRK_SSP_S_2` and `ARKODE_LSRK_SSP_S_3` were changed from 10 and 9, respectively, to their minimum allowable values of 2 and 4.

**5.10.1.3 Optional output functions**

`int LSRKStepGetNumDomEigUpdates(void *arkode_mem, long int *dom_eig_num_evals);`

Returns the number of dominant eigenvalue evaluations (so far).

**Arguments:**

- `arkode_mem` – pointer to the LSRKStep memory block.
- `dom_eig_num_evals` – number of calls to the user’s `dom_eig` function.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the LSRKStep memory was NULL

`int LSRKStepGetMaxNumStages(void *arkode_mem, int *stage_max);`

Returns the max number of stages used in any single step (so far).

**Arguments:**

- `arkode_mem` – pointer to the LSRKStep memory block.
- `stage_max` – max number of stages used.

**Return value:**

- `ARK_SUCCESS` if successful
- `ARK_MEM_NULL` if the LSRKStep memory was NULL
- `ARK_ILL_INPUT` if `stage_max` is illegal

`int LSRKStepGetNumDomEigEstRhsEvals(void *arkode_mem, long int *nfeDQ);`

Returns the number of RHS function evaluations used in the difference quotient Jacobian approximations (so far).

**Arguments:**

- `arkode_mem` – pointer to the LSRKStep memory block.
- `nfeDQ` – number of rhs calls.

**Return value:**

- `ARK_SUCCESS` if successful

- *ARK\_MEM\_NULL* if the LSRKStep memory was NULL
- *ARK\_ILL\_INPUT* if nfeDQ is illegal

Added in version 6.5.0.

#### Note

The number of RHS evaluations is non-zero only when using a dominant eigenvalue estimator and the internal Jacobian-vector product approximation.

int **LSRKStepGetNumDomEigEstIters**(void \*arkode\_mem, long int \*num\_iters);

Returns the number of iterations used in the dominant eigenvalue estimator (DEE) (so far).

#### Arguments:

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *num\_iters* – number of iterations.

#### Return value:

- *ARK\_SUCCESS* if successful
- *ARK\_MEM\_NULL* if the LSRKStep memory was NULL
- *ARK\_ILL\_INPUT* if *num\_iters* is illegal

Added in version 6.5.0.

#### 5.10.1.4 LSRKStep re-initialization function

To reinitialize the LSRKStep module for the solution of a new problem, where a prior call to *LSRKStepCreateSTS()* or *LSRKStepCreateSSP()* has been made, the user must call the function *LSRKStepReInitSTS()* or *LSRKStepReInitSSP()*, accordingly. The new problem must have the same size as the previous one. This routine retains the current settings for all LSRKStep module options and performs the same input checking and initializations that are done in *LSRKStepCreateSTS()* or *LSRKStepCreateSSP()*, but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration, and deletes any previously-set *tstop* value specified via a call to *ARKodeSetStopTime()*. Following a successful call to *LSRKStepReInitSTS()* or *LSRKStepReInitSSP()*, call *ARKodeEvolve()* again for the solution of the new problem.

One important use of the *LSRKStepReInitSTS()* and *LSRKStepReInitSSP()* function is in the treating of jump discontinuities in the RHS function. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to this routine. To stop when the location of the discontinuity is known, simply make that location a value of *tout*. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS function *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS function (communicated through *user\_data*) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **LSRKStepReInitSTS**(void \*arkode\_mem, *ARKRhsFn* rhs, *sunrealtype* t0, *N\_Vector* y0);

Provides required problem specifications and re-initializes the LSRKStep time-stepper module when using STS methods.

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *rhs* – the name of the C function (of type [ARKRhsFn\(\)](#)) defining the right-hand side function.
- *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 LSRKStep memory was NULL
- *ARK\_MEM\_FAIL* if memory allocation failed
- *ARK\_NO\_MALLOC* if memory allocation failed
- *ARK\_CONTROLLER\_ERR* if unable to reset error controller object
- *ARK\_ILL\_INPUT* if an argument had an illegal value.

**Note**

If using a [SUNDomEigEstimator](#), the initial guess for the dominant eigenvalue should be reinitialized with [SUNDomEigEstimator\\_SetInitialGuess\(\)](#).

int **LSRKStepReInitSSP**(void \*arkode\_mem, [ARKRhsFn](#) rhs, *sunrealtype* t0, *N\_Vector* y0);

Provides required problem specifications and re-initializes the LSRKStep time-stepper module when using SSP methods.

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

**Arguments:**

- *arkode\_mem* – pointer to the LSRKStep memory block.
- *rhs* – the name of the C function (of type [ARKRhsFn\(\)](#)) defining the right-hand side function.
- *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 LSRKStep memory was NULL
- *ARK\_MEM\_FAIL* if memory allocation failed
- *ARK\_NO\_MALLOC* if memory allocation failed
- *ARK\_CONTROLLER\_ERR* if unable to reset error controller object
- *ARK\_ILL\_INPUT* if an argument had an illegal value.

## 5.10.2 User-supplied functions

In addition to the required [ARKRhsFn](#) arguments that define the IVP, RKL and RKC methods additionally require an [ARKDomEigFn](#) function to estimate the dominant eigenvalue.

### 5.10.2.1 The dominant eigenvalue estimation

When running LSRKStep with either the RKC or RKL methods, the user must supply a dominant eigenvalue estimation function of type [ARKDomEigFn](#):

```
typedef int (*ARKDomEigFn)(sunrealtype t, N_Vector y, N_Vector fn, sunrealtype *lambdaR, sunrealtype *lambdaI,
void *user_data, N_Vector temp1, N_Vector temp2, N_Vector temp3);
```

These functions compute the dominant eigenvalue of the Jacobian of the ODE right-hand side for a given value of the independent variable  $t$  and state vector  $y$ .

**Param t**

the current value of the independent variable.

**Param y**

the current value of the dependent variable vector.

**Param fn**

the current value of the vector  $f(t, y)$ .

**Param lambdaR**

The real part of the dominant eigenvalue.

**Param lambdaI**

The imaginary part of the dominant eigenvalue.

**Param user\_data**

the *user\_data* pointer that was passed to [ARKodeSetUserData\(\)](#).

**Param tmp\***

pointers to memory allocated to variables of type *N\_Vector* which can be used by an ARK-DomEigFn as temporary storage or work space.

**Return**

An [ARKDomEigFn](#) should return 0 if successful and any nonzero for a failure.

## 5.11 Using the MRISStep time-stepping module

This section is concerned with the use of the MRISStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. Usage of MRISStep follows that of the rest of ARKODE, and so in this section we primarily focus on those usage aspects that are specific to MRISStep.

### 5.11.1 A skeleton of the user's main program

While MRISStep usage generally follows the same pattern as the rest of ARKODE, since it involves the solution of both MRISStep for the slow time scale and another time integrator for the fast time scale, we summarize the differences in using MRISStep here. Steps that are unchanged from the skeleton program presented in §5.2 are *italicized*.

1. *Initialize parallel or multi-threaded environment, if appropriate.*
2. *Create the SUNDIALS simulation context object*

3. *Set problem dimensions, etc.*
4. *Set vector of initial values*
5. Create an inner stepper object to solve the fast (inner) IVP

- If using an ARKODE stepper module for the fast integrator, create and configure the stepper as normal following the steps detailed in the section for the desired stepper.

Once the ARKODE stepper object is setup, create an `MRISStepInnerStepper` object with `ARKodeCreateMRISStepInnerStepper()`.

- If supplying a user-defined fast (inner) integrator, create the `MRISStepInnerStepper` object as described in section §5.11.4.

#### Note

When using `ARKStep` as a fast (inner) integrator it is the user's responsibility to create, configure, and attach the integrator to the `MRISStep` module. User-specified options regarding how this fast integration should be performed (e.g., adaptive vs. fixed time step, explicit/implicit/ImEx partitioning, algebraic solvers, etc.) will be respected during evolution of the fast time scale during `MRISStep` integration.

Due to the algorithms supported in `MRISStep`, the `ARKStep` module used for the fast time scale must be configured with an identity mass matrix.

If a `user_data` pointer needs to be passed to user functions called by the fast (inner) integrator then it should be attached here by calling `ARKodeSetUserData()`. This `user_data` pointer will only be passed to user-supplied functions that are attached to the fast (inner) integrator. To supply a `user_data` pointer to user-supplied functions called by the slow (outer) integrator the desired pointer should be attached by calling `ARKodeSetUserData()` after creating the `MRISStep` memory below. The `user_data` pointers attached to the inner and outer integrators may be the same or different depending on what is required by the user code.

Specifying a rootfinding problem for the fast integration is not supported. Rootfinding problems should be created and initialized with the slow integrator. See the steps below and `ARKodeRootInit()` for more details.

6. Create an `MRISStep` object for the slow (outer) integration

Create the `MRISStep` object by calling `MRISStepCreate()`. One of the inputs to `MRISStepCreate()` is the `MRISStepInnerStepper` object for solving the fast (inner) IVP created in the previous step.

7. If using fixed step sizes, then set the slow step size by calling `ARKodeSetFixedStep()` on the `MRISStep` object to specify the slow time step size.

If using adaptive slow steps, then specify the desired integration tolerances as normal. By default, `MRISStep` will use a “decoupled” (see §2.11.1.1) I controller (see §13.2). Alternately, create and attach a multirate temporal controller (see §13.4).

8. Create and configure implicit solvers (*as appropriate*)

Specifically, if `MRISStep` is configured with an implicit slow right-hand side function in the prior step, then the following steps are recommended:

1. *Specify integration tolerances*
2. *Create matrix object*
3. *Create linear solver object*
4. *Set linear solver optional inputs*
5. *Attach linear solver module*



6. *Create nonlinear solver object*
7. *Attach nonlinear solver module*
8. *Set nonlinear solver optional inputs*
9. *Set optional inputs*
10. *Specify rootfinding problem*
11. *Advance solution in time*
12. *Get optional outputs*
13. *Deallocate memory for solution vector*
14. *Free solver memory*
  - If ARKStep was used as the fast (inner) IVP integrator, call `MRISStepInnerStepper_Free()` and `ARKodeFree()` to free the memory allocated for the fast (inner) integrator.
  - If a user-defined fast (inner) integrator was supplied, free the integrator content and call `MRISStepInnerStepper_Free()` to free the MRISStepInnerStepper object.
  - Call `ARKodeFree()` to free the memory allocated for the MRISStep slow integration object.
15. *Free linear solver and matrix memory (as appropriate)*
16. *Free nonlinear solver memory (as appropriate)*
17. *Free the SUNContext object*
18. *Finalize MPI, if used*

### 5.11.2 MRISStep User-callable functions

This section describes the MRISStep-specific functions that may be called by the user to setup and then solve an IVP using the MRISStep time-stepping module. The large majority of these routines merely wrap *underlying ARKODE functions*, and are now deprecated – each of these are clearly marked. However, some of these user-callable functions are specific to MRISStep, as explained below.

As discussed in the main *ARKODE user-callable function introduction*, each of ARKODE's time-stepping modules clarifies the categories of user-callable functions that it supports. MRISStep supports the following categories:

- temporal adaptivity
- implicit nonlinear and/or linear solvers

MRISStep also has forcing function support when converted to a *SUNStepper* or *MRISStepInnerStepper*. See `ARKodeCreateSUNStepper()` and `ARKStepCreateMRISStepInnerStepper()` for additional details.

#### 5.11.2.1 MRISStep initialization and deallocation functions

void \***MRISStepCreate**(*ARKRhsFn* fse, *ARKRhsFn* fsi, *sunrealtype* t0, *N\_Vector* y0, *MRISStepInnerStepper* stepper, *SUNContext* sunctx)

This function allocates and initializes memory for a problem to be solved using the MRISStep time-stepping module in ARKODE.

##### Parameters

- **fse** – the name of the function (of type *ARKRhsFn*) defining the explicit slow portion of the right-hand side function in  $\dot{y} = f^E(t, y) + f^I(t, y) + f^F(t, y)$ .

- **fsi** – the name of the function (of type [ARKRhFn\(\)](#)) defining the implicit slow portion of the right-hand side function in  $\dot{y} = f^E(t, y) + f^I(t, y) + f^F(t, y)$ .
- **t0** – the initial value of  $t$ .
- **y0** – the initial condition vector  $y(t_0)$ .
- **stepper** – an [MRISStepInnerStepper](#) for integrating the fast time scale.
- **sunctx** – the [SUNContext](#) object (see §4.2)

**Returns**

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

**Example usage:**

```
/* fast (inner) and slow (outer) ARKODE objects */
void *inner_arkode_mem = NULL;
void *outer_arkode_mem = NULL;

/* MRISStepInnerStepper to wrap the inner (fast) object */
MRISStepInnerStepper stepper = NULL;

/* create an ARKODE object, setting fast (inner) right-hand side
   functions and the initial condition */
inner_arkode_mem = *StepCreate(...);

/* configure the inner integrator */
retval = ARKodeSet*(inner_arkode_mem, ...);

/* create MRISStepInnerStepper wrapper for the ARKODE integrator */
flag = ARKodeCreateMRISStepInnerStepper(inner_arkode_mem, &stepper);

/* create an MRISStep object, setting the slow (outer) right-hand side
   functions and the initial condition */
outer_arkode_mem = MRISStepCreate(fse, fsi, t0, y0, stepper, sunctx)
```

**Example codes:**

- `examples/arkode/C_serial/ark_brusselator_mri.c`
- `examples/arkode/C_serial/ark_twowaycouple_mri.c`
- `examples/arkode/C_serial/ark_brusselator_1D_mri.c`
- `examples/arkode/C_serial/ark_onewaycouple_mri.c`
- `examples/arkode/C_serial/ark_reaction_diffusion_mri.c`
- `examples/arkode/C_serial/ark_kpr_mri.c`
- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`
- `examples/arkode/CXX_serial/ark_test_kpr_nestedmri.cpp` (uses `MRISStep` within itself)

`void MRISStepFree(void **arkode_mem)`

This function frees the problem memory *arkode\_mem* created by [MRISStepCreate\(\)](#).

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.

Deprecated since version 6.1.0: Use [ARKodeFree\(\)](#) instead.

### 5.11.2.2 MRISStep tolerance specification functions

int **MRISStepSStolerances**(void \*arkode\_mem, *sunrealtype* reltol, *sunrealtype* abstol)

This function specifies scalar relative and absolute tolerances.

#### Parameters

- **arkode\_mem** – pointer to the MRISStep memory block.
- **reltol** – scalar relative tolerance.
- **abstol** – scalar absolute tolerance.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARK\_NO\_MALLOC** – if the MRISStep memory was not allocated by the time-stepping module
- **ARK\_ILL\_INPUT** – if an argument had an illegal value (e.g. a negative tolerance).

Deprecated since version 6.1.0: Use [ARKodeSStolerances\(\)](#) instead.

int **MRISStepSVtolerances**(void \*arkode\_mem, *sunrealtype* 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).

#### Parameters

- **arkode\_mem** – pointer to the MRISStep memory block.
- **reltol** – scalar relative tolerance.
- **abstol** – vector containing the absolute tolerances for each solution component.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARK\_NO\_MALLOC** – if the MRISStep memory was not allocated by the time-stepping module
- **ARK\_ILL\_INPUT** – if an argument had an illegal value (e.g. a negative tolerance).

Deprecated since version 6.1.0: Use [ARKodeSVtolerances\(\)](#) instead.

int **MRISStepWFTolerances**(void \*arkode\_mem, *ARKEwtFn* efun)

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

#### Parameters

- **arkode\_mem** – pointer to the MRISStep memory block.
- **efun** – the name of the function (of type [ARKEwtFn\(\)](#)) that implements the error weight vector computation.

#### Return values

- **ARK\_SUCCESS** – if successful

- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARK\_NO\_MALLOC** – if the MRISStep memory was not allocated by the time-stepping module

Deprecated since version 6.1.0: Use [ARKodeWftolerances\(\)](#) instead.

### 5.11.2.3 Linear solver interface functions

int **MRISStepSetLinearSolver**(void \*arkode\_mem, *SUNLinearSolver* LS, *SUNMatrix* J)

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

#### Parameters

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

#### Return values

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARKLS\_MEM\_FAIL** – if there was a memory allocation failure
- **ARKLS\_ILL\_INPUT** – if ARKLS is incompatible with the provided *LS* or *J* input objects, or the current *N\_Vector* module.

#### Note

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

If *LS* is a matrix-based linear solver, then the template Jacobian matrix *J* will be used in the solve process, so if additional storage is required within the *SUNMatrix* object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular *SUNMATRIX* type in §9 for further information).

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

Deprecated since version 6.1.0: Use [ARKodeSetLinearSolver\(\)](#) instead.

### 5.11.2.4 Nonlinear solver interface functions

int **MRISStepSetNonlinearSolver**(void \*arkode\_mem, *SUNNonlinearSolver* NLS)

This function specifies the *SUNNonlinearSolver* object that MRISStep should use for implicit stage solves.

#### Parameters

- **arkode\_mem** – pointer to the MRISStep memory block.
- **NLS** – the *SUNNonlinearSolver* object to use.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARK\_MEM\_FAIL** – if there was a memory allocation failure
- **ARK\_ILL\_INPUT** – if MRISStep is incompatible with the provided *NLS* input object.

**Note**

MRISStep will use the Newton `SUNNonlinearSolver` module by default; a call to this routine replaces that module with the supplied *NLS* object.

Deprecated since version 6.1.0: Use `ARKodeSetNonlinearSolver()` instead.

**5.11.2.5 Rootfinding initialization function**

int **MRISStepRootInit**(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 `MRISStepCreate()`, and before `MRISStepEvolve()`.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **nrtfn** – number of functions  $g_i$ , an integer  $\geq 0$ .
- **g** – name of user-supplied function, of type `ARKRootFn()`, defining the functions  $g_i$  whose roots are sought.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep 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.

**Note**

To disable the rootfinding feature after it has already been initialized, or to free memory associated with MRISStep's rootfinding module, call `MRISStepRootInit` with *nrtfn* = 0.

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

Rootfinding is only supported for the slow (outer) integrator and should not be activated for the fast (inner) integrator.

Deprecated since version 6.1.0: Use `ARKodeRootInit()` instead.

### 5.11.2.6 MRISolver solver function

int **MRISolverEvolve**(void \*arkode\_mem, *sunrealtype* tout, *N\_Vector* yout, *sunrealtype* \*tret, int itask)

Integrates the ODE over an interval in  $t$ .

#### Parameters

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

The *ARK\_NORMAL* option causes the solver to take internal steps until it has just overtaken a user-specified output time, *tout*, in the direction of integration, i.e.  $t_{n-1} < tout \leq t_n$  for forward integration, or  $t_n \leq tout < t_{n-1}$  for backward integration. It will then compute an approximation to the solution  $y(tout)$  by interpolation (as described in §2.2).

The *ARK\_ONE\_STEP* option tells the solver to only take a single internal step,  $y_{n-1} \rightarrow y_n$ , and return the solution at that point,  $y_n$ , in the vector *yout*.

#### Return values

- **ARK\_SUCCESS** – if successful.
- **ARK\_ROOT\_RETURN** – if *MRISolverEvolve()* succeeded, and found one or more roots. If the number of root functions, *nrtfn*, is greater than 1, call *ARKodeGetRootInfo()* to see which  $g_i$  were found to have a root at (*\*tret*).
- **ARK\_TSTOP\_RETURN** – if *MRISolverEvolve()* 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 *MRISolverEvolve()* 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:
  - (a) A component of the error weight vector became zero during internal time-stepping.
  - (b) The linear solver initialization function (called by the user after calling *ARKStepCreate()*) failed to set the linear solver-specific *lsolve* field in *arkode\_mem*.
  - (c) A root of one of the root functions was found both at a point  $t$  and also very near  $t$ .
- **ARK\_TOO\_MUCH\_WORK** – if the solver took *mxstep* internal steps but could not reach *tout*. The default value for *mxstep* is *MXSTEP\_DEFAULT* = 500.
- **ARK\_CONV\_FAILURE** – if convergence test failures occurred too many times (*ark\_maxncf*) during one internal time step.
- **ARK\_LINIT\_FAIL** – if the linear solver's initialization function failed.
- **ARK\_LSETUP\_FAIL** – if the linear solver's setup routine failed in an unrecoverable manner.
- **ARK\_LSOLVE\_FAIL** – if the linear solver's solve routine failed in an unrecoverable manner.
- **ARK\_VECTOROP\_ERR** – a vector operation error occurred.

- **ARK\_INNERSTEP\_FAILED** – if the inner stepper returned with an unrecoverable error. The value returned from the inner stepper can be obtained with [MRISetGetLastInnerStepFlag\(\)](#).
- **ARK\_INVALID\_TABLE** – if an invalid coupling table was provided.

**Note**

The input vector *yout* can use the same memory as the vector *y0* of initial conditions that was passed to [MRISetCreate\(\)](#).

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 [MRISetEvolve\(\)](#) failures.

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

On any error return in which one or more internal steps were taken by [MRISetEvolve\(\)](#), 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.

Deprecated since version 6.1.0: Use [ARKodeEvolve\(\)](#) instead.

### 5.11.2.7 Optional input functions

#### Optional inputs for MRISet

int [MRISetSetDefaults](#)(void \*arkode\_mem)

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

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

This function does not change problem-defining function pointers *fs* and *ff* or the *user\_data* pointer. It also does not affect any data structures or options related to root-finding (those can be reset using [MRISetRootInit\(\)](#)).

Deprecated since version 6.1.0: Use [ARKodeSetDefaults\(\)](#) instead.

int **MRISetInterpolantType**(void \*arkode\_mem, int itype)

Deprecated since version 6.1.0: This function is now a wrapper to [ARKodeSetInterpolantType\(\)](#), see the documentation for that function instead.

int **MRISetInterpolantDegree**(void \*arkode\_mem, int degree)

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

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **degree** – requested polynomial degree.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory or interpolation module are NULL
- **ARK\_INTERP\_FAIL** – if this is called after [MRISetEvolve\(\)](#)
- **ARK\_ILL\_INPUT** – if an argument has an illegal value or the interpolation module has already been initialized

#### Note

Allowed values are between 0 and 5.

This routine should be called *after* [MRISetCreate\(\)](#) and *before* [MRISetEvolve\(\)](#). After the first call to [MRISetEvolve\(\)](#) the interpolation degree may not be changed without first calling [MRISetReInit\(\)](#).

If a user calls both this routine and [MRISetSetInterpolantType\(\)](#), then [MRISetSetInterpolantType\(\)](#) must be called first.

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

Changed in version 5.5.1: When  $q = 1$ , a linear interpolant is the default to ensure values obtained by the integrator are returned at the ends of the time interval.

Deprecated since version 6.1.0: Use [ARKodeSetInterpolantDegree\(\)](#) instead.

int **MRISetDenseOrder**(void \*arkode\_mem, int dord)

Deprecated since version 5.2.0: Use [ARKodeSetInterpolantDegree\(\)](#) instead.

int **MRISetDiagnostics**(void \*arkode\_mem, FILE \*diagfp)

Specifies the file pointer for a diagnostics file where all MRISet step adaptivity and solver information is written.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **diagfp** – pointer to the diagnostics output file.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value



**Note**

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.

Deprecated since version 5.2.0: Use [SUNLogger\\_SetInfoFilename\(\)](#) instead.

int **MRISetStepSetFixedStep**(void \*arkode\_mem, *sunrealtype* hs)

Set the slow step size used within MRISetStep for the following internal step(s).

**Parameters**

- **arkode\_mem** – pointer to the MRISetStep memory block.
- **hs** – value of the outer (slow) step size.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISetStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

The step sizes used by the inner (fast) stepper may be controlled through calling the appropriate “Set” routines on the inner integrator.

Deprecated since version 6.1.0: Use [ARKodeSetFixedStep\(\)](#) instead.

int **MRISetStepSetMaxHnilWarns**(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 MRISetStep will instead return with an error.

**Parameters**

- **arkode\_mem** – pointer to the MRISetStep memory block.
- **mxhnil** – maximum allowed number of warning messages ( $> 0$ ).

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISetStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

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

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

Deprecated since version 6.1.0: Use [ARKodeSetMaxHnilWarns\(\)](#) instead.

int **MRISetMaxNumSteps**(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 MRISet will return with an error.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **mxsteps** – maximum allowed number of internal steps.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Note

Passing *mxsteps* = 0 results in MRISet using the default value (500).

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

Deprecated since version 6.1.0: Use [ARKSetMaxNumSteps\(\)](#) instead.

int **MRISetStopTime**(void \*arkode\_mem, *sunrealtype* tstop)

Specifies the value of the independent variable *t* past which the solution is not to proceed.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **tstop** – stopping time for the integrator.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Note

The default is that no stop time is imposed.

Once the integrator returns at a stop time, any future testing for **tstop** is disabled (and can be re-enabled only through a new call to [MRISetStopTime\(\)](#)).

A stop time not reached before a call to [MRISetReInit\(\)](#) or [MRISetReset\(\)](#) will remain active but can be disabled by calling [MRISetClearStopTime\(\)](#).

Deprecated since version 6.1.0: Use [ARKSetStopTime\(\)](#) instead.

int **MRISetInterpolateStopTime**(void \*arkode\_mem, *sunbooleantype* interp)

Specifies that the output solution should be interpolated when the current *t* equals the specified **tstop** (instead of merely copying the internal solution  $y_n$ ).

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.

- **interp** – flag indicating to use interpolation (1) or copy (0).

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory is NULL

Added in version 5.6.0.

Deprecated since version 6.1.0: Use [ARKodeSetInterpolateStopTime\(\)](#) instead.

int **MRISStepClearStopTime**(void \*arkode\_mem)

Disables the stop time set with [MRISStepSetStopTime\(\)](#).

#### Parameters

- **arkode\_mem** – pointer to the MRISStep memory block.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory is NULL

#### Note

The stop time can be re-enabled though a new call to [MRISStepSetStopTime\(\)](#).

Added in version 5.5.1.

Deprecated since version 6.1.0: Use [ARKodeClearStopTime\(\)](#) instead.

int **MRISStepSetUserData**(void \*arkode\_mem, void \*user\_data)

Specifies the user data block *user\_data* for the outer integrator and attaches it to the main MRISStep memory block.

#### Parameters

- **arkode\_mem** – pointer to the MRISStep memory block.
- **user\_data** – pointer to the user data.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Note

If specified, the pointer to *user\_data* is passed to all user-supplied functions called by the outer integrator for which it is an argument; otherwise NULL is passed.

To attach a user data block to the inner integrator call the appropriate *SetUserData* function for the inner integrator memory structure (e.g., [ARKStepSetUserData\(\)](#) if the inner stepper is ARKStep). This pointer may be the same as or different from the pointer attached to the outer integrator depending on what is required by the user code.

Deprecated since version 6.1.0: Use [ARKodeSetUserData\(\)](#) instead.

int **MRISetPreInnerFn**(void \*arkode\_mem, *MRISetPreInnerFn* prefn)

Specifies the function called *before* each inner integration.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **prefn** – the name of the C function (of type *MRISetPreInnerFn()*) defining pre inner integration function.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL

int **MRISetPostInnerFn**(void \*arkode\_mem, *MRISetPostInnerFn* postfn)

Specifies the function called *after* each inner integration.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **postfn** – the name of the C function (of type *MRISetPostInnerFn()*) defining post inner integration function.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL

## Optional inputs for IVP method selection

Table 5.4: Optional inputs for IVP method selection

Optional input	Function name	Default
Select the default MRI method of a given order	<i>MRISetSetOrder()</i>	3
Set MRI coupling coefficients	<i>MRISetSetCoupling()</i>	internal

int **MRISetSetOrder**(void \*arkode\_mem, int ord)

Select the default MRI method of a given order.

The default order is 3. An order less than 1 will result in using the default.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **ord** – the method order.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL

Deprecated since version 6.1.0: Use *ARKSetOrder()* instead.

int **MRISetCoupling**(void \*arkode\_mem, *MRISetCoupling* C)

Specifies a customized set of slow-to-fast coupling coefficients for the MRI method.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **C** – the table of coupling coefficients for the MRI method.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Note

For a description of the *MRISetCoupling* type and related functions for creating Butcher tables see §5.11.3. This routine will be called by *ARKSetOptions()* when using the key “arkid.coupling\_table\_name”, where C is itself constructed by passing the command-line option to *MRISetCoupling\_LoadTableByName()*.

#### Warning

This should not be used with *ARKSetOrder()*.

### Optional inputs for implicit stage solves

int **MRISetLinear**(void \*arkode\_mem, int timedepend)

Specifies that the implicit slow right-hand side function,  $f^I(t, y)$  is linear in  $y$ .

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **timedepend** – flag denoting whether the Jacobian of  $f^I(t, y)$  is time-dependent (1) or not (0). Alternately, when using a matrix-free iterative linear solver this flag denotes time dependence of the preconditioner.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Note

Tightens the linear solver tolerances and takes only a single Newton iteration. Calls *MRISetSetDeltaGammaMax()* to enforce Jacobian recomputation when the step size ratio changes by more than 100 times the unit roundoff (since nonlinear convergence is not tested). Only applicable when used in combination with the modified or inexact Newton iteration (not the fixed-point solver).

The only SUNDIALS-provided SUNNonlinearSolver module that is compatible with the *MRISetLinear()* option is the Newton solver.

Deprecated since version 6.1.0: Use *ARKodeSetLinear()* instead.

int **MRISetNonlinear**(void \*arkode\_mem)

Specifies that the implicit slow right-hand side function,  $f^I(t, y)$  is nonlinear in  $y$ .

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

This is the default behavior of MRISet, so the function is primarily useful to undo a previous call to *MRISetLinear()*. Calls *MRISetDeltaGammaMax()* to reset the step size ratio threshold to the default value.

Deprecated since version 6.1.0: Use *ARKodeSetNonlinear()* instead.

int **MRISetPredictorMethod**(void \*arkode\_mem, int method)

Specifies the method to use for predicting implicit solutions.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **method** – the predictor method
  - 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. **deprecated**

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

The default value is 0. If *method* is set to an undefined value, this default predictor will be used.

**Warning**

The “bootstrap” predictor (option 4 above) has been deprecated, and will be removed from a future release.

Deprecated since version 6.1.0: Use [ARKodeSetPredictorMethod\(\)](#) instead.

int **MRISetMaxNonlinIters**(void \*arkode\_mem, int maxcor)

Specifies the maximum number of nonlinear solver iterations permitted per slow MRI stage within each time step.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **maxcor** – maximum allowed solver iterations per stage ( $> 0$ ).

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value or if the SUNNONLINSOL module is NULL
- **ARK\_NLS\_OP\_ERR** – if the SUNNONLINSOL object returned a failure flag

**Note**

The default value is 3; set *maxcor*  $\leq 0$  to specify this default.

Deprecated since version 6.1.0: Use [ARKodeSetMaxNonlinIters\(\)](#) instead.

int **MRISetNonlinConvCoef**(void \*arkode\_mem, *sunrealtype* nlscoef)

Specifies the safety factor used within the nonlinear solver convergence test.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **nlscoef** – coefficient in nonlinear solver convergence test ( $> 0.0$ ).

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

The default value is 0.1; set *nlscoef*  $\leq 0$  to specify this default.

Deprecated since version 6.1.0: Use [ARKodeSetNonlinConvCoef\(\)](#) instead.

int **MRISetNonlinCRDown**(void \*arkode\_mem, *sunrealtype* crdown)

Specifies the constant used in estimating the nonlinear solver convergence rate.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **crdown** – nonlinear convergence rate estimation constant (default is 0.3).

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

Note
Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetNonlinCRDown\(\)](#) instead.

int **MRISetNonlinRDiv**(void \*arkode\_mem, *sunrealtype* rdiv)

Specifies the nonlinear correction threshold beyond which the iteration will be declared divergent.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **rdiv** – tolerance on nonlinear correction size ratio to declare divergence (default is 2.3).

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

Note
Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [ARKodeSetNonlinRDiv\(\)](#) instead.

int **MRISetStagePredictFn**(void \*arkode\_mem, *ARKStagePredictFn* PredictStage)

Sets the user-supplied function to update the implicit stage predictor prior to execution of the nonlinear or linear solver algorithms that compute the implicit stage solution.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **PredictStage** – name of user-supplied predictor function. If NULL, then any previously-provided stage prediction function will be disabled.

**Return values**

- **ARK\_SUCCESS** – if successful



- **ARK\_MEM\_NULL** – if the MRISStep memory is NULL

**Note**

See §5.4.6 for more information on this user-supplied routine.

Deprecated since version 6.1.0: Use [ARKodeSetStagePredictFn\(\)](#) instead.

int **MRISStepSetNlsRhsFn**(void \*arkode\_mem, *ARKRhsFn* nls\_fs)

Specifies an alternative implicit slow right-hand side function for evaluating  $f^I(t, y)$  within nonlinear system function evaluations.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **nls\_fs** – the alternative C function for computing the right-hand side function  $f^I(t, y)$  in the ODE.

**Return values**

- **ARK\_SUCCESS** – if successful.
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL.

**Note**

The default is to use the implicit slow right-hand side function provided to [MRISStepCreate\(\)](#) in nonlinear system functions. If the input implicit slow right-hand side function is NULL, the default is used.

When using a non-default nonlinear solver, this function must be called *after* [MRISStepSetNonlinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetNlsRhsFn\(\)](#) instead.

int **MRISStepSetDeduceImplicitRhs**(void \*arkode\_mem, *sunbooleantype* deduce)

Specifies if implicit stage derivatives are deduced without evaluating  $f^I$ . See §2.15.1 for more details.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **deduce** – If SUNFALSE (default), the stage derivative is obtained by evaluating  $f^I$  with the stage solution returned from the nonlinear solver. If SUNTRUE, the stage derivative is deduced without an additional evaluation of  $f^I$ .

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory is NULL

Added in version 5.2.0.

Deprecated since version 6.1.0: Use [ARKodeSetDeduceImplicitRhs\(\)](#) instead.

## Linear solver interface optional input functions

### Optional inputs for the ARKLS linear solver interface

int **MRISetDeltaGammaMax**(void \*arkode\_mem, *sunrealtype* dgmax)

Specifies a scaled step size ratio tolerance, beyond which the linear solver setup routine will be signaled.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **dgmax** – tolerance on step size ratio change before calling linear solver setup routine (default is 0.2).

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Note

Any non-positive parameter will imply a reset to the default value.

Deprecated since version 6.1.0: Use [\*ARKSetDeltaGammaMax\(\)\*](#) instead.

int **MRISetLSetupFrequency**(void \*arkode\_mem, int msbp)

Specifies the frequency of calls to the linear solver setup routine.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **msbp** – the linear solver setup frequency.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory is NULL

#### Note

Positive values of **msbp** specify the linear solver setup frequency. For example, an input of 1 means the setup function will be called every time step while an input of 2 means it will be called every other time step. If **msbp** is 0, the default value of 20 will be used. A negative value forces a linear solver step at each implicit stage.

Deprecated since version 6.1.0: Use [\*ARKSetLSetupFrequency\(\)\*](#) instead.

int **MRISetJacEvalFrequency**(void \*arkode\_mem, long int msbj)

Specifies the frequency for recomputing the Jacobian or recommending a preconditioner update.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **msbj** – the Jacobian re-computation or preconditioner update frequency.

**Return values**

- **ARKLS\_SUCCESS** – if successful.
- **ARKLS\_MEM\_NULL** – if the MRISep memory was NULL.
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL.

**Note**

The Jacobian update frequency is only checked *within* calls to the linear solver setup routine, as such values of  $msbj < msbp$  will result in recomputing the Jacobian every  $msbp$  steps. See [MRISepSetLSetupFrequency\(\)](#) for setting the linear solver setup frequency  $msbp$ .

Passing a value  $msbj \leq 0$  indicates to use the default value of 50.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [MRISepSetLinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetJacEvalFrequency\(\)](#) instead.

**Optional inputs for matrix-based SUNLinearSolver modules**

int **MRISepSetJacFn**(void \*arkode\_mem, [ARKLSJacFn](#) jac)

Specifies the Jacobian approximation routine to be used for the matrix-based solver with the ARKLS interface.

**Parameters**

- **arkode\_mem** – pointer to the MRISep memory block.
- **jac** – name of user-supplied Jacobian approximation function.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This routine must be called after the ARKLS linear solver interface has been initialized through a call to [MRISepSetLinearSolver\(\)](#).

By default, ARKLS uses an internal difference quotient function for dense and band matrices. If NULL is passed in for *jac*, this default is used. An error will occur if no *jac* is supplied when using other matrix types.

The function type [ARKLSJacFn\(\)](#) is described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetJacFn\(\)](#) instead.

int **MRISepSetLinSysFn**(void \*arkode\_mem, [ARKLSLinSysFn](#) linsys)

Specifies the linear system approximation routine to be used for the matrix-based solver with the ARKLS interface.

**Parameters**

- **arkode\_mem** – pointer to the MRISep memory block.

- **linsys** – name of user-supplied linear system approximation function.

#### Return values

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

#### Note

This routine must be called after the ARKLS linear solver interface has been initialized through a call to [MRISepSetLinearSolver\(\)](#).

By default, ARKLS uses an internal linear system function that leverages the SUNMATRIX API to form the system  $I - \gamma J$ . If NULL is passed in for *linsys*, this default is used.

The function type [ARKLSLinSysFn\(\)](#) is described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetLinSysFn\(\)](#) instead.

int **MRISepSetLinearSolutionScaling**(void \*arkode\_mem, *sunbooleantype* onoff)

Enables or disables scaling the linear system solution to account for a change in  $\gamma$  in the linear system. For more details see §10.2.1.

#### Parameters

- **arkode\_mem** – pointer to the MRISep memory block.
- **onoff** – flag to enable (SUNTRUE) or disable (SUNFALSE) scaling

#### Return values

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISep memory was NULL
- **ARKLS\_ILL\_INPUT** – if the attached linear solver is not matrix-based

#### Note

Linear solution scaling is enabled by default when a matrix-based linear solver is attached.

Deprecated since version 6.1.0: Use [ARKodeSetLinearSolutionScaling\(\)](#) instead.

### Optional inputs for matrix-free SUNLinearSolver modules

int **MRISepSetJacTimes**(void \*arkode\_mem, [ARKLSJacTimesSetupFn](#) jtsetup, [ARKLSJacTimesVecFn](#) jtimes)

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

#### Parameters

- **arkode\_mem** – pointer to the MRISep memory block.
- **jtsetup** – user-defined Jacobian-vector setup function. Pass NULL if no setup is necessary.
- **jtimes** – user-defined Jacobian-vector product function.

#### Return values

- **ARKLS\_SUCCESS** – if successful.
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL.
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – if an input has an illegal value.
- **ARKLS\_SUNLS\_FAIL** – if an error occurred when setting up the Jacobian-vector product in the SUNLinearSolver object used by the ARKLS interface.

**Note**

The default is to use an internal finite difference quotient for *jt看imes* and to leave out *jtsetup*. If NULL is passed to *jt看imes*, these defaults are used. A user may specify non-NULL *jt看imes* and NULL *jtsetup* inputs.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [MRISStepSetLinearSolver\(\)](#).

The function types [ARKLSJacTimesSetupFn](#) and [ARKLSJacTimesVecFn](#) are described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetJacTimes\(\)](#) instead.

int **MRISStepSetJacTimesRhsFn**(void \*arkode\_mem, [ARKRhsFn](#) jt看imesRhsFn)

Specifies an alternative implicit right-hand side function for use in the internal Jacobian-vector product difference quotient approximation.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **jt看imesRhsFn** – the name of the C function defining the alternative right-hand side function.

**Return values**

- **ARKLS\_SUCCESS** – if successful.
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL.
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – if an input has an illegal value.

**Note**

The default is to use the implicit right-hand side function provided to [MRISStepCreate\(\)](#) in the internal difference quotient. If the input implicit right-hand side function is NULL, the default is used.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [MRISStepSetLinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetJacTimesRhsFn\(\)](#) instead.

**Optional inputs for iterative SUNLinearSolver modules**

int **MRISStepSetPreconditioner**(void \*arkode\_mem, [ARKLPrecSetupFn](#) psetup, [ARKLPrecSolveFn](#) psolve)

Specifies the user-supplied preconditioner setup and solve functions.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **psetup** – user defined preconditioner setup function. Pass NULL if no setup is needed.
- **psolve** – user-defined preconditioner solve function.

**Return values**

- **ARKLS\_SUCCESS** – if successful.
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL.
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – if an input has an illegal value.
- **ARKLS\_SUNLS\_FAIL** – if an error occurred when setting up preconditioning in the SUNLinearSolver object used by the ARKLS interface.

**Note**

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

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [MRISStepSetLinearSolver\(\)](#).

Both of the function types [ARKLSPrecSetupFn\(\)](#) and [ARKLSPrecSolveFn\(\)](#) are described in §5.4.

Deprecated since version 6.1.0: Use [ARKodeSetPreconditioner\(\)](#) instead.

int **MRISStepSetEpsLin**(void \*arkode\_mem, *sunrealtype* eplifac)

Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the linear iteration.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **eplifac** – linear convergence safety factor.

**Return values**

- **ARKLS\_SUCCESS** – if successful.
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL.
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL.
- **ARKLS\_ILL\_INPUT** – if an input has an illegal value.

**Note**

Passing a value  $eplifac \leq 0$  indicates to use the default value of 0.05.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [MRISStepSetLinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetEpsLin\(\)](#) instead.

int **MRISStepSetLSNormFactor**(void \*arkode\_mem, *sunrealtype* nrmfac)

Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for Newton linear system solves e.g.,  $tol_{L2} = fac * tol_{WRMS}$ .

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **nrmfac** – the norm conversion factor. If *nrmfac* is:
  - > 0 then the provided value is used.
  - = 0 then the conversion factor is computed using the vector length i.e.,  $\text{nrmfac} = \sqrt{\text{N\_VGetLength}(y)}$  (*default*).
  - < 0 then the conversion factor is computed using the vector dot product i.e.,  $\text{nrmfac} = \sqrt{\text{N\_VDotProd}(v, v)}$  where all the entries of *v* are one.

**Return values**

- **ARK\_SUCCESS** – if successful.
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL.

**Note**

This function must be called *after* the ARKLS system solver interface has been initialized through a call to [MRISStepSetLinearSolver\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetLSNormFactor\(\)](#) instead.

**Rootfinding optional input functions**

int **MRISStepSetRootDirection**(void \*arkode\_mem, int \*rootdir)

Specifies the direction of zero-crossings to be located and returned.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **rootdir** – state array of length *nrtfn*, the number of root functions  $g_i$  (the value of *nrtfn* was supplied in the call to [MRISStepRootInit\(\)](#)). If  $\text{rootdir}[i] == 0$  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 values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

The default behavior is to monitor for both zero-crossing directions.

Deprecated since version 6.1.0: Use [ARKodeSetRootDirection\(\)](#) instead.

int **MRISStepSetNoInactiveRootWarn**(void \*arkode\_mem)

Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

**Parameters**

- **arkode\_mem** – pointer to the MRISep memory block.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISep memory is NULL

#### Note

MRISep 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), MRISep will issue a warning which can be disabled with this optional input function.

Deprecated since version 6.1.0: Use [ARKodeSetNoInactiveRootWarn\(\)](#) instead.

### 5.11.2.8 Interpolated output function

int **MRISepGetDky**(void \*arkode\_mem, *sunrealtype* t, int k, *N\_Vector* dky)

Computes the  $k$ -th derivative of the function  $y$  at the time  $t$ , i.e.  $y^{(k)}(t)$ , 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. This routine uses an interpolating polynomial of degree  $\min(\text{degree}, 5)$ , where *degree* is the argument provided to [MRISepSetInterpolantDegree\(\)](#). The user may request  $k$  in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$  where  $kmax$  depends on the choice of interpolation module. For Hermite interpolants  $kmax = 5$  and for Lagrange interpolants  $kmax = 3$ .

#### Parameters

- **arkode\_mem** – pointer to the MRISep 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 values

- **ARK\_SUCCESS** – if successful
- **ARK\_BAD\_K** – if  $k$  is not in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$ .
- **ARK\_BAD\_T** – if  $t$  is not in the interval  $[t_n - h_n, t_n]$
- **ARK\_BAD\_DKY** – if the *dky* vector was NULL
- **ARK\_MEM\_NULL** – if the MRISep memory is NULL

#### Note

It is only legal to call this function after a successful return from [MRISepEvolve\(\)](#).

A user may access the values  $t_n$  and  $h_n$  via the functions [MRISepGetCurrentTime\(\)](#) and [MRISepGetLastStep\(\)](#), respectively.

Deprecated since version 6.1.0: Use [ARKodeGetDky\(\)](#) instead.



### 5.11.2.9 Optional output functions

#### Main solver optional output functions

int **MRISetGetNumInnerStepperFails**(void \*arkode\_mem, long int \*inner\_fails)

Returns the number of recoverable failures reported by the inner stepper (so far).

##### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **inner\_fails** – number of failed fast (inner) integrations.

##### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

Added in version 6.2.0.

int **MRISetGetWorkSpace**(void \*arkode\_mem, long int \*lenrw, long int \*leniw)

Returns the MRISet real and integer workspace sizes.

##### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **lenrw** – the number of `realtype` values in the MRISet workspace.
- **leniw** – the number of integer values in the MRISet workspace.

##### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetWorkSpace\(\)](#) instead.

int **MRISetGetNumSteps**(void \*arkode\_mem, long int \*nssteps, long int \*nfsteps)

Returns the cumulative number of slow and fast internal steps taken by the solver (so far).

##### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **nssteps** – number of slow steps taken in the solver.
- **nfsteps** – number of fast steps taken in the solver.

##### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumSteps\(\)](#) instead.

int **MRISetGetLastStep**(void \*arkode\_mem, *sunrealtype* \*hlast)

Returns the integration step size taken on the last successful internal step.

##### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **hlast** – step size taken on the last internal step.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetLastStep\(\)](#) instead.

int **MRISepGetCurrentTime**(void \*arkode\_mem, *sunrealtype* \*tcur)

Returns the current internal time reached by the solver.

**Parameters**

- **arkode\_mem** – pointer to the MRISep memory block.
- **tcur** – current internal time reached.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentTime\(\)](#) instead.

int **MRISepGetCurrentState**(void \*arkode\_mem, *N\_Vector* \*ycur)

Returns the current internal solution reached by the solver.

**Parameters**

- **arkode\_mem** – pointer to the MRISep memory block.
- **ycur** – current internal solution.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISep memory was NULL

**Note**

Users should exercise extreme caution when using this function, as altering values of *ycur* may lead to undesirable behavior, depending on the particular use case and on when this routine is called.

Deprecated since version 6.1.0: Use [ARKodeGetCurrentState\(\)](#) instead.

int **MRISepGetCurrentGamma**(void \*arkode\_mem, *sunrealtype* \*gamma)

Returns the current internal value of  $\gamma$  used in the implicit solver Newton matrix (see equation (2.47)).

**Parameters**

- **arkode\_mem** – pointer to the MRISep memory block.
- **gamma** – current step size scaling factor in the Newton system.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentGamma\(\)](#) instead.

int **MRISetGetTolScaleFactor**(void \*arkode\_mem, *sunrealtype* \*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.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **tolsfac** – suggested scaling factor for user-supplied tolerances.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

Deprecated since version 6.1.0: Use [ARKSetGetTolScaleFactor\(\)](#) instead.

int **MRISetGetErrWeights**(void \*arkode\_mem, *N\_Vector* eweight)

Returns the current error weight vector.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **eweight** – solution error weights at the current time.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

**Note**

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

Deprecated since version 6.1.0: Use [ARKSetGetErrWeights\(\)](#) instead.

int **MRISetPrintAllStats**(void \*arkode\_mem, FILE \*outfile, *SUNOutputFormat* fmt)

Outputs all of the integrator, nonlinear solver, linear solver, and other statistics.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **outfile** – pointer to output file.
- **fmt** – the output format:
  - [SUN\\_OUTPUTFORMAT\\_TABLE](#) – prints a table of values
  - [SUN\\_OUTPUTFORMAT\\_CSV](#) – prints a comma-separated list of key and value pairs e.g., key1,value1,key2,value2,...

**Return values**

- **ARK\_SUCCESS** – if the output was successfully.
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL.
- **ARK\_ILL\_INPUT** – if an invalid formatting option was provided.

**Note**

The Python module `tools/suntools` provides utilities to read and output the data from a SUNDIALS CSV output file using the key and value pair format.

Added in version 5.2.0.

Deprecated since version 6.1.0: Use [`ARKodePrintAllStats\(\)`](#) instead.

char \***MRISetGetReturnFlagName**(long int flag)

Returns the name of the MRISet constant corresponding to *flag*. See [\*ARKODE Constants\*](#).

**Parameters**

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

**Returns**

A string containing the name of the corresponding constant.

**Warning**

The user is responsible for freeing the returned string.

Deprecated since version 6.1.0: Use [`ARKodeGetReturnFlagName\(\)`](#) instead.

int **MRISetGetNumRhsEvals**(void \*arkode\_mem, long int \*nfse\_evals, long int \*nfsi\_evals)

Returns the number of calls to the user's outer (slow) right-hand side functions,  $f^E$  and  $f^I$ , so far.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **nfse\_evals** – number of calls to the user's  $f^E(t, y)$  function.
- **nfsi\_evals** – number of calls to the user's  $f^I(t, y)$  function.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

Deprecated since version 6.2.0: Use [`ARKodeGetNumRhsEvals\(\)`](#) instead.

int **MRISetGetNumStepSolveFails**(void \*arkode\_mem, long int \*ncnf)

Returns the number of failed steps due to a nonlinear solver failure (so far).

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **ncnf** – number of step failures.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

Deprecated since version 6.1.0: Use [`ARKodeGetNumStepSolveFails\(\)`](#) instead.

int **MRISetGetCurrentCoupling**(void \*arkode\_mem, *MRISetCoupling* \*C)

Returns the MRI coupling table currently in use by the solver.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **C** – pointer to slow-to-fast MRI coupling structure.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

**Note**

The *MRISetCoupling* data structure is defined in the header file `arkode/arkode_mriset.h`. For more details see §5.11.3.

int **MRISetGetLastInnerStepFlag**(void \*arkode\_mem, int \*flag)

Returns the last return value from the inner stepper.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **flag** – inner stepper return value.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

int **MRISetGetUserData**(void \*arkode\_mem, void \*\*user\_data)

Returns the user data pointer previously set with *MRISetSetUserData()*.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **user\_data** – memory reference to a user data pointer

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the ARKStep memory was NULL

Added in version 5.3.0.

Deprecated since version 6.1.0: Use *ARKodeGetUserData()* instead.

## Implicit solver optional output functions

int **MRISetGetNumLinSolvSetups**(void \*arkode\_mem, long int \*nlinsetups)

Returns the number of calls made to the linear solver's setup routine (so far).

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.

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

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL

**Note**

This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumLinSolvSetups\(\)`](#) instead.

int **MRISStepGetNumNonlinSolvIters**(void \*arkode\_mem, long int \*nniters)

Returns the number of nonlinear solver iterations performed (so far).

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **nniters** – number of nonlinear iterations performed.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARK\_NLS\_OP\_ERR** – if the SUNNONLINSOL object returned a failure flag

**Note**

This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumNonlinSolvIters\(\)`](#) instead.

int **MRISStepGetNumNonlinSolvConvFails**(void \*arkode\_mem, long int \*nncfails)

Returns the number of nonlinear solver convergence failures that have occurred (so far).

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **nncfails** – number of nonlinear convergence failures.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL

**Note**

This is only accumulated for the “life” of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [`ARKodeGetNumNonlinSolvConvFails\(\)`](#) instead.

int **MRISStepGetNonlinSolvStats**(void \*arkode\_mem, long int \*nniters, long int \*nncfails)

Returns all of the nonlinear solver statistics in a single call.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **nniters** – number of nonlinear iterations performed.
- **nncfails** – number of nonlinear convergence failures.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARK\_NLS\_OP\_ERR** – if the SUNNONLINSOL object returned a failure flag

**Note**

These are only accumulated for the “life” of the nonlinear solver object; the counters are reset whenever a new nonlinear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNonlinSolvStats\(\)](#) instead.

**Rootfinding optional output functions**

int **MRISStepGetRootInfo**(void \*arkode\_mem, int \*rootsfound)

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

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **rootsfound** – array of length *nrtfn* with the indices of the user functions  $g_i$  found to have a root (the value of *nrtfn* was supplied in the call to [MRISStepRootInit\(\)](#)). For  $i = 0 \dots nrtfn-1$ , **rootsfound**[*i*] is nonzero if  $g_i$  has a root, and 0 if not.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL

**Note**

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  $g_i$ .

Deprecated since version 6.1.0: Use [ARKodeGetRootInfo\(\)](#) instead.

int **MRISStepGetNumGEvals**(void \*arkode\_mem, long int \*ngevals)

Returns the cumulative number of calls made to the user’s root function  $g$ .

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **ngevals** – number of calls made to *g* so far.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumGEvals\(\)](#) instead.

**Linear solver interface optional output functions**

int **MRISStepGetJac**(void \*arkode\_mem, *SUNMatrix* \*J)

Returns the internally stored copy of the Jacobian matrix of the ODE implicit slow right-hand side function.

**Parameters**

- **arkode\_mem** – the MRISStep memory structure
- **J** – the Jacobian matrix

**Return values**

- **ARKLS\_SUCCESS** – the output value has been successfully set
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized

**Warning**

This function is provided for debugging purposes and the values in the returned matrix should not be altered.

Deprecated since version 6.1.0: Use [ARKodeGetJac\(\)](#) instead.

int **MRISStepGetJacTime**(void \*arkode\_mem, *sunrealtype* \*t\_J)

Returns the time at which the internally stored copy of the Jacobian matrix of the ODE implicit slow right-hand side function was evaluated.

**Parameters**

- **arkode\_mem** – the MRISStep memory structure
- **t\_J** – the time at which the Jacobian was evaluated

**Return values**

- **ARKLS\_SUCCESS** – the output value has been successfully set
- **ARKLS\_MEM\_NULL** – arkode\_mem was NULL
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized

Deprecated since version 6.1.0: Use [ARKodeGetJacTime\(\)](#) instead.

int **MRISStepGetJacNumSteps**(void \*arkode\_mem, long int \*nst\_J)

Returns the value of the internal step counter at which the internally stored copy of the Jacobian matrix of the ODE implicit slow right-hand side function was evaluated.

**Parameters**



- **arkode\_mem** – the MRISStep memory structure
- **nst\_J** – the value of the internal step counter at which the Jacobian was evaluated

**Return values**

- **ARKLS\_SUCCESS** – the output value has been successfully set
- **ARKLS\_MEM\_NULL** – **arkode\_mem** was NULL
- **ARKLS\_LMEM\_NULL** – the linear solver interface has not been initialized

Deprecated since version 6.1.0: Use [ARKodeGetJacNumSteps\(\)](#) instead.

int **MRISStepGetLinWorkSpace**(void \*arkode\_mem, long int \*lenrwLS, long int \*leniwLS)

Returns the real and integer workspace used by the ARKLS linear solver interface.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **lenrwLS** – the number of `realtype` values in the ARKLS workspace.
- **leniwLS** – the number of integer values in the ARKLS workspace.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the `SUNLinearSolver` object attached to it. The template Jacobian matrix allocated by the user outside of ARKLS is not included in this report.

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

Deprecated since version 6.1.0: Use [ARKodeGetLinWorkSpace\(\)](#) instead.

int **MRISStepGetNumJacEvals**(void \*arkode\_mem, long int \*njevals)

Returns the number of Jacobian evaluations.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **njevals** – number of Jacobian evaluations.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumJacEvals\(\)](#) instead.

int **MRISStepGetNumPrecEvals**(void \*arkode\_mem, long int \*npevals)

Returns the total number of preconditioner evaluations, i.e., the number of calls made to *psetup* with *jok* = SUNFALSE and that returned \*jcurPtr = SUNTRUE.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **npevals** – the current number of calls to *psetup*.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumPrecEvals\(\)](#) instead.

int **MRISStepGetNumPrecSolves**(void \*arkode\_mem, long int \*npsolves)

Returns the number of calls made to the preconditioner solve function, *psolve*.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **npsolves** – the number of calls to *psolve*.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumPrecSolves\(\)](#) instead.

int **MRISetGetNumLinIters**(void \*arkode\_mem, long int \*nliters)

Returns the cumulative number of linear iterations.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **nliters** – the current number of linear iterations.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISet memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISet, or when MRISet is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumLinIters\(\)](#) instead.

int **MRISetGetNumLinConvFails**(void \*arkode\_mem, long int \*nlcfails)

Returns the cumulative number of linear convergence failures.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **nlcfails** – the current number of linear convergence failures.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISet memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISet, or when MRISet is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumLinConvFails\(\)](#) instead.

int **MRISetGetNumJTSetupEvals**(void \*arkode\_mem, long int \*njtsetup)

Returns the cumulative number of calls made to the user-supplied Jacobian-vector setup function, *jtsetup*.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **njtsetup** – the current number of calls to *jtsetup*.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISet memory was NULL

- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumJTSetupEvals\(\)](#) instead.

int **MRISStepGetNumJtimesEvals**(void \*arkode\_mem, long int \*njvevals)

Returns the cumulative number of calls made to the Jacobian-vector product function, *jtimes*.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **njvevals** – the current number of calls to *jtimes*.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumJtimesEvals\(\)](#) instead.

int **MRISStepGetNumLinRhsEvals**(void \*arkode\_mem, long int \*nfevalsLS)

Returns the number of calls to the user-supplied implicit right-hand side function  $f^I$  for finite difference Jacobian or Jacobian-vector product approximation.

**Parameters**

- **arkode\_mem** – pointer to the MRISStep memory block.
- **nfevalsLS** – the number of calls to the user implicit right-hand side function.

**Return values**

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

**Note**

The value *nfevalsLS* is incremented only if the default internal difference quotient function is used.

This is only accumulated for the “life” of the linear solver object; the counter is reset whenever a new linear solver module is “attached” to MRISStep, or when MRISStep is resized.

Deprecated since version 6.1.0: Use [ARKodeGetNumLinRhsEvals\(\)](#) instead.

int **MRIStepGetLastLinFlag**(void \*arkode\_mem, long int \*lsflag)

Returns the last return value from an ARKLS routine.

#### Parameters

- **arkode\_mem** – pointer to the MRIStep memory block.
- **lsflag** – the value of the last return flag from an ARKLS function.

#### Return values

- **ARKLS\_SUCCESS** – if successful
- **ARKLS\_MEM\_NULL** – if the MRIStep memory was NULL
- **ARKLS\_LMEM\_NULL** – if the linear solver memory was NULL

#### Note

If the ARKLS setup function failed when using the SUNLINSOL\_DENSE or SUNLINSOL\_BAND modules, then the value of *lsflag* is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix. For all other failures, *lsflag* is negative.

Otherwise, if the ARKLS setup function failed (*MRIStepEvolve()* returned *ARK\_LSETUP\_FAIL*), then *lsflag* will be *SUNLS\_PSET\_FAIL\_UNREC*, *SUNLS\_ASET\_FAIL\_UNREC* or *SUNLS\_PACKAGE\_FAIL\_UNREC*.

If the ARKLS solve function failed (*MRIStepEvolve()* returned *ARK\_LSOLVE\_FAIL*), then *lsflag* contains the error return flag from the SUNLinearSolver object, which will be one of:

- *SUNLS\_MEM\_NULL*, indicating that the SUNLinearSolver memory is NULL;
- *SUNLS\_ATIMES\_NULL*, indicating that a matrix-free iterative solver was provided, but is missing a routine for the matrix-vector product approximation,
- *SUNLS\_ATIMES\_FAIL\_UNREC*, indicating an unrecoverable failure in the *Jv* function;
- *SUNLS\_PSOLVE\_NULL*, indicating that an iterative linear solver was configured to use preconditioning, but no preconditioner solve routine was provided,
- *SUNLS\_PSOLVE\_FAIL\_UNREC*, indicating that the preconditioner solve function failed unrecoverably;
- *SUNLS\_GS\_FAIL*, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only);
- *SUNLS\_QRSOL\_FAIL*, indicating that the matrix *R* was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or
- *SUNLS\_PACKAGE\_FAIL\_UNREC*, indicating an unrecoverable failure in an external iterative linear solver package.

Deprecated since version 6.1.0: Use *ARKodeGetLastLinFlag()* instead.

char \***MRIStepGetLinReturnFlagName**(long int lsflag)

Returns the name of the ARKLS constant corresponding to *lsflag*.

#### Parameters

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

#### Returns

The return value is a string containing the name of the corresponding constant. If using the

SUNLINSOL\_DENSE or SUNLINSOL\_BAND modules, then if  $1 \leq lsflag \leq n$  (LU factorization failed), this routine returns “NONE”.

**Warning**

The user is responsible for freeing the returned string.

Deprecated since version 6.1.0: Use [ARKodeGetLinReturnFlagName\(\)](#) instead.

**General usability functions**

int **MRISetWriteParameters**(void \*arkode\_mem, FILE \*fp)

Outputs all MRISet solver parameters to the provided file pointer.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **fp** – pointer to use for printing the solver parameters.

**Return values**

- **ARKS\_SUCCESS** – if successful
- **ARKS\_MEM\_NULL** – if the MRISet memory was NULL

**Note**

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.

Deprecated since version 6.1.0: Use [ARKodeWriteParameters\(\)](#) instead.

int **MRISetWriteCoupling**(void \*arkode\_mem, FILE \*fp)

Outputs the current MRI coupling table to the provided file pointer.

**Parameters**

- **arkode\_mem** – pointer to the MRISet memory block.
- **fp** – pointer to use for printing the Butcher tables.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL

**Note**

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

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

Deprecated since version 6.1.0: Use `MRISetGetCurrentCoupling()` and `MRISetCoupling_Write()` instead.

### 5.11.2.10 MRISet re-initialization function

To reinitialize the MRISet module for the solution of a new problem, where a prior call to `MRISetCreate()` has been made, the user must call the function `MRISetReInit()`. The new problem must have the same size as the previous one. This routine retains the current settings for all MRISet module options and performs the same input checking and initializations that are done in `MRISetCreate()`, but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration, and deletes any previously-set `tstop` value specified via a call to `MRISetSetStopTime()`. Following a successful call to `MRISetReInit()`, call `MRISetEvolve()` again for the solution of the new problem.

The use of `MRISetReInit()` requires that the number of Runge–Kutta stages for both the slow and fast methods be no larger for the new problem than for the previous problem.

One important use of the `MRISetReInit()` function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to this routine. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **MRISetReInit**(void \*arkode\_mem, *ARKRhsFn* fse, *ARKRhsFn* fsi, *sunrealtype* t0, *N\_Vector* y0)

Provides required problem specifications and re-initializes the MRISet outer (slow) stepper.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **fse** – the name of the function (of type *ARKRhsFn*) defining the explicit slow portion of the right-hand side function in  $\dot{y} = f^E(t, y) + f^I(t, y) + f^F(t, y)$ .
- **fsi** – the name of the function (of type *ARKRhsFn*) defining the implicit slow portion of the right-hand side function in  $\dot{y} = f^E(t, y) + f^I(t, y) + f^F(t, y)$ .
- **t0** – the initial value of  $t$ .
- **y0** – the initial condition vector  $y(t_0)$ .

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL
- **ARK\_MEM\_FAIL** – if a memory allocation failed
- **ARK\_ILL\_INPUT** – if an argument has an illegal value.

#### Note

If the inner (fast) stepper also needs to be reinitialized, its reinitialization function should be called before calling `MRISetReInit()` to reinitialize the outer stepper.

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, `MRISetReInit()` also sends an error message to the error handler function.

### 5.11.2.11 MRISet reset function

int **MRISetReset**(void \*arkode\_mem, *sunrealtype* tR, *N\_Vector* yR)

Resets the current MRISet outer (slow) time-stepper module state to the provided independent variable value and dependent variable vector.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **tR** – the value of the independent variable  $t$ .
- **yR** – the value of the dependent variable vector  $y(t_R)$ .

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISet memory was NULL
- **ARK\_MEM\_FAIL** – if a memory allocation failed
- **ARK\_ILL\_INPUT** – if an argument has an illegal value.

#### Note

If the inner (fast) stepper also needs to be reset, its reset function should be called before calling `MRISetReset()` to reset the outer stepper.

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, `MRISetReset()` also sends an error message to the error handler function.

Changed in version 5.3.0: This now calls the corresponding `MRISetInnerResetFn` with the same ( $t_R$ ,  $y_R$ ) arguments for the `MRISetInnerStepper` object that is used to evolve the MRI “fast” time scale subproblems.

Deprecated since version 6.1.0: Use `ARKodeReset()` instead.

### 5.11.2.12 MRISet system resize function

int **MRISetResize**(void \*arkode\_mem, *N\_Vector* yR, *sunrealtype* tR, *ARKVecResizeFn* resize, void \*resize\_data)

Re-initializes MRISet with a different state vector.

#### Parameters

- **arkode\_mem** – pointer to the MRISet memory block.
- **yR** – the newly-sized solution vector, holding the current dependent variable values  $y(t_R)$ .
- **tR** – the current value of the independent variable  $t_R$  (this must be consistent with  $y_R$ ).
- **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 MRISet vectors.

#### Return values



- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the MRISStep memory was NULL
- **ARK\_NO\_MALLOC** – if *arkode\_mem* was not allocated.
- **ARK\_ILL\_INPUT** – if an argument has an illegal value.

**Note**

If an error occurred, *MRISStepResize()* 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 *MRISStepResize()*. Moreover, the existing ARKLS interface should then be deleted and recreated by attaching the updated SUNLinearSolver (and possibly SUNMatrix) object(s) through calls to *MRISStepSetLinearSolver()*.

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 *MRISStepResize()*, so the new absolute tolerance vector should be re-set **following** each call to *MRISStepResize()* through a new call to *MRISStepSVtolerances()*.

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

**Example codes:**

For an example showing usage of the similar *ARKStepResize()* routine, see the supplied serial C example problem, *ark\_heat1D\_adapt.c*.

Deprecated since version 6.1.0: Use *ARKodeResize()* instead.

### 5.11.3 MRI Coupling Coefficients Data Structure

MRISStep supplies several built-in MIS, MRI-GARK, and IMEX-MRI-GARK methods, see §5.11.3.2 for the current set of coupling tables and their corresponding identifiers. Additionally, a user may supply a custom set of slow-to-fast time scale coupling coefficients by constructing a coupling table and attaching it with *MRISStepSetCoupling()*. A given MRI coupling table can encode any of the MRI methods supported by MRISStep. The family of MRI method encoded by the table is determined by an enumerated type, *MRISTEP\_METHOD\_TYPE*:

enum **MRISTEP\_METHOD\_TYPE**

The MRI method family encoded by a *MRISStepCoupling* table

enumerator **MRISTEP\_EXPLICIT**

An explicit MRI-GARK method (does not support a slow implicit operator,  $f^I$ ).

enumerator **MRISTEP\_IMPLICIT**

An implicit MRI-GARK method (does not support a slow explicit operator,  $f^E$ ).

enumerator **MRISTEP\_IMEX**

An IMEX-MRK-GARK method.

enumerator **MRISTEP\_MERK**

A explicit MERK method (does not support a slow implicit operator,  $f^I$ ).

enumerator **MRISTEP\_SR**

An IMEX-MRI-SR method.

The MRI coupling tables themselves are stored in an *MRISTepCoupling()* object which is a pointer to a *MRISTepCouplingMem* structure:

```
typedef MRISTepCouplingMem *MRISTepCoupling
```

```
struct MRISTepCouplingMem
```

Structure for storing the coupling coefficients defining an MIS, MRI-GARK, or IMEX-MRI-GARK method.

As described in §2.7, the coupling from the slow time scale to the fast time scale is encoded by a vector of slow stage time abscissae,  $c^S \in \mathbb{R}^{s+1}$  and a set of coupling tensors  $\Gamma \in \mathbb{R}^{(s+1) \times (s+1) \times k}$  and  $\Omega \in \mathbb{R}^{(s+1) \times (s+1) \times k}$ .

*MRISTEP\_METHOD\_TYPE* **type**

Flag indicating the type of MRI method encoded by this table.

int **nmat**

The value of  $k$  above i.e., number of coupling matrices in  $\Omega$  for the slow-nonstiff terms and/or in  $\Gamma$  for the slow-stiff terms in (2.11).

int **stages**

The number of abscissae i.e.,  $s + 1$  above.

int **q**

The method order of accuracy.

int **p**

The embedding order of accuracy.

*sunrealtype* \***c**

An array of length [stages] containing the slow abscissae  $c^S$  for the method.

*sunrealtype* \*\*\***W**

A three-dimensional array with dimensions [nmat] [stages+1] [stages] containing the method's  $\Omega$  coupling coefficients for the slow-nonstiff (explicit) terms in (2.11).

*sunrealtype* \*\*\***G**

A three-dimensional array with dimensions [nmat] [stages+1] [stages] containing the method's  $\Gamma$  coupling coefficients for the slow-stiff (implicit) terms in (2.11).

int **ngroup**

Number of stage groups for the method (only relevant for MERK methods).

int \*\***group**

A two-dimensional array with dimensions [stages] [stages] that encodes which stages should be combined together within fast integration groups (only relevant for MERK methods).

### 5.11.3.1 MRISStepCoupling functions

This section describes the functions for creating and interacting with coupling tables. The function prototypes and as well as the relevant integer constants are defined `arkode/arkode_mristep.h`.

Table 5.5: MRISStepCoupling functions

Function name	Description
<i>MRISStepCoupling_LoadTable()</i>	Loads a pre-defined MRISStepCoupling table by ID
<i>MRISStepCoupling_LoadTableByName()</i>	Loads a pre-defined MRISStepCoupling table by name
<i>MRISStepCoupling_Alloc()</i>	Allocate an empty MRISStepCoupling table
<i>MRISStepCoupling_Create()</i>	Create a new MRISStepCoupling table from coefficients
<i>MRISStepCoupling_MISToMRI()</i>	Create a new MRISStepCoupling table from a Butcher table
<i>MRISStepCoupling_Copy()</i>	Create a copy of a MRISStepCoupling table
<i>MRISStepCoupling_Space()</i>	Get the MRISStepCoupling table real and integer workspace sizes
<i>MRISStepCoupling_Free()</i>	Deallocate a MRISStepCoupling table
<i>MRISStepCoupling_Write()</i>	Write the MRISStepCoupling table to an output file

*MRISStepCoupling* **MRISStepCoupling\_LoadTable**(*ARKODE\_MRITableID* method)

Retrieves a specified coupling table. For further information on the current set of coupling tables and their corresponding identifiers, see §5.11.3.2.

#### Parameters

- **method** – the coupling table identifier.

#### Returns

An *MRISStepCoupling* structure if successful. A NULL pointer if *method* was invalid or an allocation error occurred.

*MRISStepCoupling* **MRISStepCoupling\_LoadTableByName**(const char \*method)

Retrieves a specified coupling table. For further information on the current set of coupling tables and their corresponding name, see §5.11.3.2.

#### Parameters

- **method** – the coupling table name.

#### Returns

An *MRISStepCoupling* structure if successful. A NULL pointer if *method* was invalid, *method* was "ARKODE\_MRI\_NONE", or an allocation error occurred.

#### Note

This function is case sensitive.

*MRISStepCoupling* **MRISStepCoupling\_Alloc**(int nmat, int stages, *MRISTEP\_METHOD\_TYPE* type)

Allocates an empty MRISStepCoupling table.

#### Parameters

- **nmat** – the value of  $k$  i.e., number of number of coupling matrices in  $\Omega$  for the slow-nonstiff terms and/or in  $\Gamma$  for the slow-stiff terms in (2.11).
- **stages** – number of stages in the coupling table.
- **type** – the type of MRI method the table will encode.

**Returns**

An *MRISStepCoupling* structure if successful. A NULL pointer if *stages* or *type* was invalid or an allocation error occurred.

**Note**

For *MRISTEP\_EXPLICIT* tables, the *G* and *group* arrays are not allocated.

For *MRISTEP\_IMPLICIT* tables, the *W* and *group* arrays are not allocated.

For *MRISTEP\_IMEX* tables, the *group* array is not allocated.

For *MRISTEP\_MERK* tables, the *G* array is not allocated.

For *MRISTEP\_SR* tables, the *group* array is not allocated.

When allocated, both  $\Omega$  and  $\Gamma$  are initialized to all zeros, so only nonzero coefficients need to be provided.

When allocated, all entries in *group* are initialized to -1, indicating an unused group and/or the end of a stage group. Users who supply a custom *MRISTEP\_MERK* table should overwrite all active stages in each group. For example the ARKODE\_MERK32 method has 4 stages that are evolved in 3 groups – the first group consists of stage 1, the second group consists of stages 2 and 4, while the third group consists of stage 3. Thus *ngroup* should equal 3, and *group* should have non-default entries

```
C->group[0][0] = 1;
C->group[1][0] = 2;
C->group[1][1] = 4;
C->group[2][0] = 3;
```

Changed in version 6.2.0: This function now supports a broader range of MRI method types.

*MRISStepCoupling* **MRISStepCoupling\_Create**(int nmat, int stages, int q, int p, *sunrealtype* \*W, *sunrealtype* \*G, *sunrealtype* \*c)

Allocates a coupling table and fills it with the given values.

This routine can only be used to create coupling tables with type *MRISTEP\_EXPLICIT*, *MRISTEP\_IMPLICIT*, or *MRISTEP\_IMEX*. The routine determines the relevant type based on whether either of the arguments *W* and *G* are NULL. Users who wish to create MRI methods of type *MRISTEP\_MERK* or *MRISTEP\_SR* must currently do so manually.

The assumed size of the input arrays *W* and *G* depends on the input value for the embedding order of accuracy, *p*.

- Non-embedded methods should be indicated by an input  $p=0$ , in which case *W* and/or *G* should have entries stored as a 1D array of size  $\text{nmat} * \text{stages} * \text{stages}$ , in row-major order.
- Embedded methods should be indicated by an input  $p>0$ , in which case *W* and/or *G* should have entries stored as a 1D array of size  $\text{nmat} * (\text{stages}+1) * \text{stages}$ , in row-major order. The additional “row” is assumed to hold the embedding coefficients.

**Parameters**

- **nmat** – the value of  $k$  i.e., number of number of coupling matrices in  $\Omega$  for the slow-nonstiff terms and/or in  $\Gamma$  for the slow-stiff terms in (2.11).
- **stages** – number of stages in the method.
- **q** – global order of accuracy for the method.
- **p** – global order of accuracy for the embedded method.

- **W** – array of values defining the explicit coupling coefficients  $\Omega$ . If the slow method is implicit pass NULL.
- **G** – array of values defining the implicit coupling coefficients  $\Gamma$ . If the slow method is explicit pass NULL.
- **c** – array of slow abscissae for the MRI method. The entries should be stored as a 1D array of length `stages`.

### Returns

An *MRISStepCoupling* structure if successful. A NULL pointer if `stages` was invalid, an allocation error occurred, or the input data arrays are inconsistent with the method type.

*MRISStepCoupling* **MRISStepCoupling\_MISToMRI**(*ARKodeButcherTable* B, int q, int p)

Creates an MRI coupling table for a traditional MIS method based on the slow Butcher table *B*.

The *s*-stage slow Butcher table must have an explicit first stage (i.e.,  $c_1 = 0$  and  $A_{1,j} = 0$  for  $1 \leq j \leq s$ ), sorted abscissae (i.e.,  $c_i \geq c_{i-1}$  for  $2 \leq i \leq s$ ), and a final abscissa value  $c_s \leq 1$ . In this case, the  $(s + 1)$ -stage coupling table is computed as

$$\Omega_{i,j,1} \text{ or } \Gamma_{i,j,1} = \begin{cases} 0, & \text{if } i = 1, \\ A_{i,j} - A_{i-1,j}, & \text{if } 2 \leq i \leq s, \\ b_j - A_{s,j}, & \text{if } i = s + 1. \end{cases}$$

and the embedding coefficients (if applicable) are computed as

$$\tilde{\Omega}_{i,j,1} \text{ or } \tilde{\Gamma}_{i,j,1} = \tilde{b}_j - A_{s,j}.$$

We note that only one of  $\Omega$  or  $\Gamma$  will be filled in. If *B* corresponded to an explicit method, then this routine fills  $\Omega$ ; if *B* is diagonally-implicit, then this routine inserts redundant “padding” stages to ensure a solve-decoupled structure and then uses the above formula to fill  $\Gamma$ .

For general slow tables with at least second-order accuracy, the MIS method will be second order. However, if the slow table is at least third order and additionally satisfies

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

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

As a result, the values of *q* and *p* may differ from the method and embedding orders of accuracy for the Runge–Kutta method encoded in *B*, which is why these arguments should be supplied separately.

If  $p > 0$  is input, then the table *B* must include embedding coefficients.

### Parameters

- **B** – the *ARKodeButcherTable* for the “slow” MIS method.
- **q** – the overall order of the MIS/MRI method.
- **p** – the overall order of the MIS/MRI embedding.

### Returns

An *MRISStepCoupling* structure if successful. A NULL pointer if an allocation error occurred.

*MRISStepCoupling* **MRISStepCoupling\_Copy**(*MRISStepCoupling* C)

Creates copy of the given coupling table.

### Parameters

- **C** – the coupling table to copy.

**Returns**

An *MRISStepCoupling* structure if successful. A NULL pointer if an allocation error occurred.

void **MRISStepCoupling\_Space**(*MRISStepCoupling* C, *sunindextype* \*liw, *sunindextype* \*lrw)

Get the real and integer workspace size for a coupling table.

**Parameters**

- **C** – the coupling table.
- **lenrw** – the number of *sunrealtype* values in the coupling table workspace.
- **leniw** – the number of integer values in the coupling table workspace.

**Return values**

- **ARK\_SUCCESS** – if successful.
- **ARK\_MEM\_NULL** – if the Butcher table memory was NULL.

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

void **MRISStepCoupling\_Free**(*MRISStepCoupling* C)

Deallocate the coupling table memory.

**Parameters**

- **C** – the coupling table.

void **MRISStepCoupling\_Write**(*MRISStepCoupling* C, FILE \*outfile)

Write the coupling table to the provided file pointer.

**Parameters**

- **C** – the coupling table.
- **outfile** – pointer to use for printing the table.

**Note**

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

### 5.11.3.2 MRI Coupling Tables

MRISStep currently includes three classes of coupling tables: those that encode methods that are explicit at the slow time scale, those that are diagonally-implicit and solve-decoupled at the slow time scale, and those that encode methods with an implicit-explicit method at the slow time scale. We list the current identifiers, multirate order of accuracy, and relevant references for each in the tables below. For methods with an implicit component, we also list the number of implicit solves per step that are required at the slow time scale.

Each of the coupling tables that are packaged with MRISStep are specified by a unique ID having type:

typedef int **ARKODE\_MRITableID**

with values specified for each method below (e.g., `ARKODE_MIS_KW3`).

Table 5.6: Explicit MRIStep coupling tables.

Table name	Method Order	Embedding Order	Slow Calls	RHS	Reference
ARKODE_MRI_GARK_FORWARD_EULER	1*	–	1		
ARKODE_MRI_GARK_ERK22a	2	1	2		[93]
ARKODE_MRI_GARK_ERK22b	2* <sup>o</sup>	1	2		[93]
ARKODE_MRI_GARK_RALSTON2	2	1	2		[88]
ARKODE_MERK21	2	1	2		[80]
ARKODE_MIS_KW3	3*	–	3		[96]
ARKODE_MRI_GARK_ERK33a	3 <sup>o</sup>	2	3		[93]
ARKODE_MRI_GARK_RALSTON3	3	2	3		[88]
ARKODE_MERK32	3	2	3		[80]
ARKODE_MRI_GARK_ERK45a	4* <sup>o</sup>	3	5		[93]
ARKODE_MERK43	4	3	6		[80]
ARKODE_MERK54	5 <sup>A</sup>	4	10		[80]

Notes regarding the above table:

1. The default method for each order when using fixed step sizes is marked with an asterisk (\*).
2. The default method for each order when using adaptive time stepping is marked with a circle (°).
3. The “Slow RHS Calls” column corresponds to the number of calls to the slow right-hand side function,  $f^E$ , per time step.
4. Note A: although all MERK methods were derived in [80] under an assumption that the fast function  $f^F(t, y)$  is linear in  $y$ , in [46] it was proven that MERK methods also satisfy all nonlinear order conditions up through their linear order. The lone exception is ARKODE\_MERK54, where it was only proven to satisfy all nonlinear conditions up to order 4 (since [46] did not establish the formulas for the order 5 conditions). All our numerical tests to date have shown ARKODE\_MERK54 to achieve fifth order for nonlinear problems, and so we conjecture that it also satisfies the nonlinear fifth order conditions.

Table 5.7: Diagonally-implicit, solve-decoupled MRI-GARK coupling tables. The default method for each order when using fixed step sizes is marked with an asterisk (\*); the default method for each order when using adaptive time stepping is marked with a circle (°). The “Implicit Solves” column corresponds to the number of slow implicit (non)linear solves required per time step.

Table name	Method Order	Embedding Order	Implicit Solves	Reference
ARKODE_MRI_GARK_BACKWARD_EULER	1* <sup>o</sup>	–	1	
ARKODE_MRI_GARK_IRK21a	2* <sup>o</sup>	1	1	[93]
ARKODE_MRI_GARK_IMPLICIT_MID-POINT	2	–	2	
ARKODE_MRI_GARK_ESDIRK34a	3* <sup>o</sup>	2	3	[93]
ARKODE_MRI_GARK_ESDIRK46a	4* <sup>o</sup>	3	5	[93]

Table 5.8: Diagonally-implicit, solve-decoupled IMEX-MRI-GARK coupling tables. The default method for each order when using fixed step sizes is marked with an asterisk (\*); the default method for each order when using adaptive time stepping is marked with a circle (°). The “Implicit Solves” column corresponds to the number of slow implicit (non)linear solves required per time step.

Table name	Method Order	Embedding Order	Implicit Solves	Reference
ARKODE_IMEX_MRI_GARK_EULER	1*	–	1	
ARKODE_IMEX_MRI_GARK_TRAPEZOIDAL	2*	–	1	
ARKODE_IMEX_MRI_GARK_MIDPOINT	2	–	2	
ARKODE_IMEX_MRI_SR21	2°	1	3	[46]
ARKODE_IMEX_MRI_GARK3a	3*	–	2	[30]
ARKODE_IMEX_MRI_GARK3b	3	–	2	[30]
ARKODE_IMEX_MRI_SR32	3°	2	4	[46]
ARKODE_IMEX_MRI_GARK4	4*	–	5	[30]
ARKODE_IMEX_MRI_SR43	4°	3	5	[46]

### 5.11.4 MRIStep Custom Inner Steppers

Recall that infinitesimal multirate methods require solving a set of auxiliary IVPs

$$\dot{v}(t) = f^F(t, v) + r_i(t), \quad v(t_{i,0}) = v_{i,0}, \quad (5.2)$$

on intervals  $t \in [t_{i,0}, t_{i,f}]$ . For the MIS, MRI-GARK and IMEX-MRI-GARK methods implemented in MRIStep, the forcing term  $r_i(t)$  presented in §2.7 can be equivalently written as

$$r_i(t) = \sum_{k \geq 1} \hat{\omega}_{i,k} \tau^{k-1} + \sum_{k \geq 1} \hat{\gamma}_{i,k} \tau^{k-1} \quad (5.3)$$

where  $\tau = (t - t_{n,i-1}^S) / (h^S \Delta c_i^S)$  is the normalized time with  $\Delta c_i^S = (c_i^S - c_{i-1}^S)$ , the slow stage times are  $t_{n,i-1}^S = t_{n-1} + c_{i-1}^S h^S$ , and the polynomial coefficient vectors are

$$\hat{\omega}_{i,k} = \frac{1}{\Delta c_i^S} \sum_{j=1}^{i-1} \Omega_{i,j,k} f^E(t_{n,j}^S, z_j) \quad \text{and} \quad \hat{\gamma}_{i,k} = \frac{1}{\Delta c_i^S} \sum_{j=1}^i \Gamma_{i,j,k} f^I(t_{n,j}^S, z_j). \quad (5.4)$$

The MERK and IMEX-MRI-SR methods included in MRIStep compute the forcing polynomial (5.3) similarly, with appropriate modifications to  $\Delta c_i^S$ ,  $t_{n,i-1}^S$ , and the coefficients (5.4).

To evolve the IVP (5.2) MRIStep utilizes a generic time integrator interface defined by the *MRIStepInnerStepper* base class. This section presents the *MRIStepInnerStepper* base class and methods that define the integrator interface as well as detailing the steps for creating an *MRIStepInnerStepper*.

#### 5.11.4.1 The MRIStepInnerStepper Class

As with other SUNDIALS classes, the *MRIStepInnerStepper* abstract base class is implemented using a C structure containing a content pointer to the derived class member data and a structure of function pointers (vtable) to the derived class implementations of the base class virtual methods.



**type *MRISStepInnerStepper***

An object for solving the fast (inner) ODE in an MRI method.

The actual definitions of the structure and the corresponding operations structure are kept private to allow for the object internals to change without impacting user code. The following sections describe the base (§5.11.4.1) and virtual methods (§5.11.4.1) that a must be provided by a derived class.

**Base Class Methods**

This section describes methods provided by the *MRISStepInnerStepper* abstract base class that aid the user in implementing derived classes. This includes functions for creating and destroying a generic base class object, attaching and retrieving the derived class content pointer, setting function pointers to derived class method implementations, and accessing base class data e.g., for computing the forcing term (5.3).

**Creating and Destroying an Object**

int **MRISStepInnerStepper\_Create**(*SUNContext* sunctx, *MRISStepInnerStepper* \*stepper)

This function creates an *MRISStepInnerStepper* object to which a user should attach the member data (content) pointer and method function pointers.

**Parameters**

- **sunctx** – the SUNDIALS simulation context.
- **stepper** – a pointer to an inner stepper object.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_FAIL** – if a memory allocation error occurs

**Example usage:**

```
/* create an instance of the base class */
MRISStepInnerStepper inner_stepper = NULL;
flag = MRISStepInnerStepper_Create(&inner_stepper);
```

**Example codes:**

- examples/arkode/CXX\_parallel/ark\_diffusion\_reaction\_p.cpp

**Note**

See §5.11.4.1 and §5.11.4.1 for details on how to attach member data and method function pointers.

int **MRISStepInnerStepper\_CreateFromSUNStepper**(*SUNStepper* sunstepper, *MRISStepInnerStepper* \*stepper)

This utility function wraps a *SUNStepper* as an *MRISStepInnerStepper*.

**Parameters**

- **sunctx** – the SUNDIALS simulation context.
- **sunstepper** – the c:type:*SUNStepper* to wrap.
- **stepper** – a pointer to an MRI inner stepper object.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_FAIL** – if a memory allocation error occurs

**Example usage:**

```
SUNStepper sunstepper = NULL;
SUNStepper_Create(ctx, &sunstepper);
/* Attach content and functions to the SUNStepper... */

MRISetInnerStepper inner_stepper = NULL;
flag = MRISetInnerStepper_CreateFromSUNStepper(sunstepper, &inner_stepper);
```

Added in version 6.2.0.

int **MRISetInnerStepper\_Free**(*MRISetInnerStepper* \*stepper)

This function destroys an *MRISetInnerStepper* object.

**Parameters**

- **stepper** – a pointer to an inner stepper object.

**Return values**

**ARK\_SUCCESS** – if successful

**Example usage:**

```
/* destroy an instance of the base class */
flag = MRISetInnerStepper_Free(&inner_stepper);
```

**Example codes:**

- examples/arkode/CXX\_parallel/ark\_diffusion\_reaction\_p.cpp

**Note**

This function only frees memory allocated within the base class and the base class structure itself. The user is responsible for freeing any memory allocated for the member data (content).

**Attaching and Accessing the Content Pointer**

int **MRISetInnerStepper\_SetContent**(*MRISetInnerStepper* stepper, void \*content)

This function attaches a member data (content) pointer to an *MRISetInnerStepper* object.

**Parameters**

- **stepper** – an inner stepper object.
- **content** – a pointer to the stepper member data.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

**Example usage:**

```
/* set the inner stepper content pointer */
MyStepperContent my_object_data;
flag = MRISetInnerStepper_SetContent(inner_stepper, &my_object_data);
```

**Example codes:**

- examples/arkode/CXX\_parallel/ark\_diffusion\_reaction\_p.cpp

int **MRISetInnerStepper\_GetContent**(*MRISetInnerStepper* stepper, void \*\*content)

This function retrieves the member data (content) pointer from an *MRISetInnerStepper* object.

**Parameters**

- **stepper** – an inner stepper object.
- **content** – a pointer to set to the stepper member data pointer.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

**Example usage:**

```
/* get the inner stepper content pointer */
void *content;
MyStepperContent *my_object_data;

flag = MRISetInnerStepper_GetContent(inner_stepper, &content);
my_object_data = (MyStepperContent*) content;
```

**Example codes:**

- examples/arkode/CXX\_parallel/ark\_diffusion\_reaction\_p.cpp

**Setting Member Functions**

int **MRISetInnerStepper\_SetEvolveFn**(*MRISetInnerStepper* stepper, *MRISetInnerEvolveFn* fn)

This function attaches an *MRISetInnerEvolveFn* function to an *MRISetInnerStepper* object.

**Parameters**

- **stepper** – an inner stepper object.
- **fn** – the *MRISetInnerStepper* function to attach.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

**Example usage:**

```
/* set the inner stepper evolve function */
flag = MRISetInnerStepper_SetEvolveFn(inner_stepper, MyEvolve);
```

**Example codes:**

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

int **MRISetInnerStepper\_SetFullRhsFn**(*MRISetInnerStepper* stepper, *MRISetInnerFullRhsFn* fn)

This function attaches an *MRISetInnerFullRhsFn* function to an *MRISetInnerStepper* object.

#### Parameters

- **stepper** – an inner stepper object.
- **fn** – the *MRISetInnerFullRhsFn* function to attach.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

#### Example usage:

```
/* set the inner stepper full right-hand side function */
flag = MRISetInnerStepper_SetFullRhsFn(inner_stepper, MyFullRHS);
```

#### Example codes:

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

int **MRISetInnerStepper\_SetResetFn**(*MRISetInnerStepper* stepper, *MRISetInnerResetFn* fn)

This function attaches an *MRISetInnerResetFn* function to an *MRISetInnerStepper* object.

#### Parameters

- **stepper** – an inner stepper object.
- **fn** – the *MRISetInnerResetFn* function to attach.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

#### Example usage:

```
/* set the inner stepper reset function */
flag = MRISetInnerStepper_SetResetFn(inner_stepper, MyReset);
```

#### Example codes:

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

int **MRISetInnerStepper\_SetAccumulatedErrorGetFn**(*MRISetInnerStepper* stepper,  
*MRISetInnerGetAccumulatedError* fn)

This function attaches an *MRISetInnerGetAccumulatedError* function to an *MRISetInnerStepper* object.

#### Parameters

- **stepper** – an inner stepper object.
- **fn** – the *MRISetInnerGetAccumulatedError* function to attach.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

int **MRISetInnerStepper\_SetAccumulatedErrorResetFn**(*MRISetInnerStepper* stepper, *MRISetInnerResetAccumulatedError* fn)

This function attaches an *MRISetInnerResetAccumulatedError* function to an *MRISetInnerStepper* object.

#### Parameters

- **stepper** – an inner stepper object.
- **fn** – the *MRISetInnerResetAccumulatedError* function to attach.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

int **MRISetInnerStepper\_SetRTolFn**(*MRISetInnerStepper* stepper, *MRISetInnerSetRTol* fn)

This function attaches an *MRISetInnerSetRTol* function to an *MRISetInnerStepper* object.

#### Parameters

- **stepper** – an inner stepper object.
- **fn** – the *MRISetInnerSetRTol* function to attach.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

## Applying and Accessing Forcing Data

When integrating the ODE (5.2) the *MRISetInnerStepper* is responsible for evaluating ODE right-hand side function  $f^F(t, v)$  as well as computing and applying the forcing term (5.3) to obtain the full right-hand side of the inner (fast) ODE (5.2). The functions in this section can be used to either apply the inner (fast) forcing or access the data necessary to construct the inner (fast) forcing polynomial. While the first of these is less intrusive and may be used to package an existing black-box IVP solver as an *MRISetInnerStepper*, the latter may be more computationally efficient since it does not traverse the data directly.

int **MRISetInnerStepper\_AddForcing**(*MRISetInnerStepper* stepper, *sunrealtype* t, *N\_Vector* ff)

This function computes the forcing term (5.3) at the input time  $t$  and adds it to input vector  $ff$ , i.e., the inner (fast) right-hand side vector.

#### Parameters

- **stepper** – an inner stepper object.
- **t** – the time at which the forcing should be evaluated.
- **f** – the vector to which the forcing should be applied.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

Example codes:

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

int **MRISStepInnerStepper\_GetForcingData**(*MRISStepInnerStepper* stepper, *sunrealtype* \*tshift, *sunrealtype* \*tscale, *N\_Vector* \*\*forcing, int \*nforcing)

This function provides access to data necessary to compute the forcing term (5.3). This includes the shift and scaling factors for the normalized time  $\tau = (t - t_{n,i-1}^S) / (h^S \Delta c_i^S)$  and the array of polynomial coefficient vectors  $\hat{\gamma}^{i,k}$ .

#### Parameters

- **stepper** – an inner stepper object.
- **tshift** – the time shift to apply to the current time when computing the forcing,  $t_{n,i-1}^S$ .
- **tscale** – the time scaling to apply to the current time when computing the forcing,  $h^S \Delta c_i^S$ .
- **forcing** – a pointer to an array of forcing vectors,  $\hat{\gamma}_{i,k}$ .
- **nforcing** – the number of forcing vectors.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_ILL\_INPUT** – if the stepper is NULL

#### Example usage:

```
int      k, flag;
int      nforcing_vecs;  /* number of forcing vectors */
double   tshift, tscale; /* time normalization values */
double   tau;           /* normalized time */
double   tau_k;         /* tau raised to the power k */
N_Vector *forcing_vecs; /* array of forcing vectors */

/* get the forcing data from the inner (fast) stepper */
flag = MRISStepInnerStepper_GetForcingData(inner_stepper, &tshift, &tscale,
                                           &forcing_vecs, &nforcing_vecs);

/* compute the normalized time, initialize tau^k */
tau = (t - tshift) / tscale;
tau_k = 1.0;

/* compute the polynomial forcing terms and add them to fast RHS vector */
for (k = 0; k < nforcing_vecs; k++)
{
    N_VLinearSum(1.0, f_fast, tau_k, forcing_vecs[k], f_fast);
    tau_k *= tau;
}
```

#### Example codes:

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

## Implementation Specific Methods

This section describes the required and optional virtual methods defined by the *MRISStepInnerStepper* abstract base class.

### Required Member Functions

An *MRISStepInnerStepper* must provide implementations of the following member functions:

```
typedef int (*MRISStepInnerEvolveFn)(MRISStepInnerStepper stepper, sunrealtype t0, sunrealtype tout, N_Vector v)
```

This function advances the state vector  $v$  for the inner (fast) ODE system from time  $t0$  to time  $tout$ .

#### Arguments:

- *stepper* – the inner stepper object.
- $t0$  – the initial time for the inner (fast) integration.
- $tout$  – the final time for the inner (fast) integration.
- $v$  – on input the state at time  $t0$  and, on output, the state at time  $tout$ .

#### Return value:

An *MRISStepInnerEvolveFn* should return 0 if successful, a positive value if a recoverable error occurred (i.e., the function could be successful if called over a smaller time interval  $[t0, tout]$ ), or a negative value if it failed unrecoverably.

#### Example codes:

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

### Optional Member Functions

An *MRISStepInnerStepper* may provide implementations of any of the following member functions:

```
typedef int (*MRISStepInnerFullRhsFn)(MRISStepInnerStepper stepper, sunrealtype t, N_Vector v, N_Vector f, int mode)
```

This function computes the full right-hand side function of the inner (fast) ODE,  $f^F(t, v)$  in (5.2) for a given value of the independent variable  $t$  and state vector  $y$ . We note that this routine should *not* include contributions from the forcing term (5.3).

#### Arguments:

- *stepper* – the inner stepper object.
- $t$  – the current value of the independent variable.
- $y$  – the current value of the dependent variable vector.
- $f$  – the output vector that forms a portion the ODE right-hand side,  $f^F(t, y)$  in (2.11).
- *mode* – a flag indicating the purpose for which the right-hand side function evaluation is called.
  - ARK\_FULLRHS\_START – called at the beginning of the simulation
  - ARK\_FULLRHS\_END – called at the end of a successful step
  - ARK\_FULLRHS\_OTHER – called elsewhere e.g., for dense output

#### Return value:

An *MRISStepInnerFullRhsFn* should return 0 if successful, or a nonzero value upon failure.

**Example codes:**

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

Changed in version v5.7.0: Supplying a full right-hand side function was made optional.

```
typedef int (*MRISetInnerResetFn)(MRISetInnerStepper stepper, sunrealtype tR, N_Vector vR)
```

This function resets the inner (fast) stepper state to the provided independent variable value and dependent variable vector.

If provided, the `MRISetInnerResetFn` function will be called *before* a call to `MRISetInnerEvolveFn` when the state was updated at the slow timescale.

**Arguments:**

- *stepper* – the inner stepper object.
- *tR* – the value of the independent variable  $t_R$ .
- *vR* – the value of the dependent variable vector  $v(t_R)$ .

**Return value:**

An `MRISetInnerResetFn` should return 0 if successful, or a nonzero value upon failure.

**Example codes:**

- `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp`

```
typedef int (*MRISetInnerGetAccumulatedError)(MRISetInnerStepper stepper, sunrealtype *accum_error)
```

This function returns an estimate of the accumulated solution error arising from the inner stepper. Both the `MRISetInnerGetAccumulatedError` and `MRISetInnerResetAccumulatedError` functions should be provided, or not; if only one is provided then MRISet will disable multirate temporal adaptivity and call neither.

**Arguments:**

- *stepper* – the inner stepper object.
- *accum\_error* – estimation of the accumulated solution error.

**Return value:**

An `MRISetInnerGetAccumulatedError` should return 0 if successful, a positive value if a recoverable error occurred (i.e., the function could be successful if called over a smaller time interval  $[t_0, t_{out}]$ ), or a negative value if it failed unrecoverably.

**Note**

This function is required when multirate temporal adaptivity has been enabled, using a `SUNAdaptController` module having type `SUN_ADAPTCONTROLLER_MRI_H_TOL`.

If provided, the `MRISetInnerGetAccumulatedError` function will always be called *after* a preceding call to the `MRISetInnerResetAccumulatedError` function.

```
typedef int (*MRISetInnerResetAccumulatedError)(MRISetInnerStepper stepper)
```

This function resets the inner stepper's accumulated solution error to zero. This function performs a different role within MRISet than the `MRISetInnerResetFn`, and thus an implementation should make no assumptions about the frequency/ordering of calls to either.

**Arguments:**

- *stepper* – the inner stepper object.



**Return value:**

An *MRISetInnerResetAccumulatedError* should return 0 if successful, or a nonzero value upon failure.

**Note**

This function is required when multirate temporal adaptivity has been enabled, using a *SUNAdaptController* module having type *SUN\_ADAPTCONTROLLER\_MRI\_H\_TOL*.

The *MRISetInnerResetAccumulatedError* function will always be called *before* any calls to the *MRISetInnerGetAccumulatedError* function.

Both the *MRISetInnerGetAccumulatedError* and *MRISetInnerResetAccumulatedError* functions should be provided, or not; if only one is provided then MRISet will disable multirate temporal adaptivity and call neither.

```
typedef int (*MRISetInnerSetRTol)(MRISetInnerStepper stepper, sunrealtype rtol)
```

This function accepts a relative tolerance for the inner stepper to use in its upcoming adaptive solve. It is assumed that if the inner stepper supports absolute tolerances as well, then these have been set up directly by the user to indicate the “noise” level for solution components.

**Arguments:**

- *stepper* – the inner stepper object.
- *rtol* – relative tolerance to use on the upcoming solve.

**Return value:**

An *MRISetInnerSetRTol* should return 0 if successful, or a nonzero value upon failure.

**Note**

This function is required when multirate temporal adaptivity has been enabled using a *SUNAdaptController* module having type *SUN\_ADAPTCONTROLLER\_MRI\_H\_TOL*.

**5.11.4.2 Implementing an MRISetInnerStepper**

To create an MRISetInnerStepper implementation:

1. Define the stepper-specific content.

This is typically a user-defined structure in C codes, a user-defined class or structure in C++ codes, or a user-defined module in Fortran codes. This content should hold any data necessary to perform the operations defined by the *MRISetInnerStepper* member functions.

2. Define implementations of the required member functions (see §5.11.4.1).

These are typically user-defined functions in C, member functions of the user-defined structure or class in C++, or functions contained in the user-defined module in Fortran.

Note that all member functions are passed the *MRISetInnerStepper* object and the stepper-specific content can, if necessary, be retrieved using *MRISetInnerStepper\_GetContent()*.

3. In the user code, before creating the MRISet memory structure with *MRISetCreate()*, do the following:
  1. Create an *MRISetInnerStepper* object with *MRISetInnerStepper\_Create()*.

2. Attach a pointer to the stepper content to the `MRIStepInnerStepper` object with `MRIStepInnerStepper_SetContent()` if necessary, e.g., when the content is a C structure.
3. Attach the member function implementations using the functions described in §5.11.4.1.
4. Attach the `MRIStepInnerStepper` object to the MRIStep memory structure with `MRIStepCreate()`.

For an example of creating and attaching a user-defined inner stepper see the example code `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp` where CVODE is wrapped as an `MRIStepInnerStepper`.

## 5.12 Using the SplittingStep time-stepping module

This section is concerned with the use of the SplittingStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. Usage of SplittingStep follows that of the rest of ARKODE, and so in this section we primarily focus on those usage aspects that are specific to SplittingStep.

### 5.12.1 A skeleton of the user's main program

While SplittingStep usage generally follows the same pattern as the rest of ARKODE, since it is the composition of other steppers, we summarize the differences in using SplittingStep here. Steps that are unchanged from the skeleton program presented in §5.2 are *italicized*.

1. *Initialize parallel or multi-threaded environment, if appropriate.*
2. *Create the SUNDIALS simulation context object*
3. *Set problem dimensions, etc.*
4. *Set vector of initial values*
5. Create a stepper object for each problem partition
  - If using an ARKODE stepper module as an partition integrator, create and configure the stepper as normal following the steps detailed in the section for the desired stepper.

Once the ARKODE stepper object is setup, create a `SUNStepper` object with `ARKodeCreateSUNStepper()`.

  - If supplying a user-defined partition integrator, create the `SUNStepper` object as described in section §14.2.

#### Note

When using ARKODE for partition integrators, it is the user's responsibility to create and configure the integrator. User-specified options regarding how the integration should be performed (e.g., adaptive vs. fixed time step, explicit/implicit/ImEx partitioning, algebraic solvers, etc.) will be respected during evolution of a partition during SplittingStep integration.

If a `user_data` pointer needs to be passed to user functions called by a partition integrator then it should be attached to the partition integrator here by calling `ARKodeSetUserData()`. This `user_data` pointer will only be passed to user-supplied functions that are attached to a partition integrator. To supply a `user_data` pointer to user-supplied functions called by the SplittingStep integrator, the desired pointer should be attached by calling `ARKodeSetUserData()` after creating the SplittingStep memory below. The `user_data` pointers attached to the partition and SplittingStep integrators may be the same or different depending on what is required by the user code.

Specifying a rootfinding problem for a partition integrator is not supported. Rootfinding problems should be created and initialized with SplittingStep. See the steps below and `ARKodeRootInit()` for more details.

## 6. Create a SplittingStep object

Create the SplittingStep object by calling *SplittingStepCreate()*. One of the inputs to *SplittingStepCreate()* is an array of *SUNStepper* objects with one to evolve each partition.

## 7. Set the SplittingStep step size

Call *ARKodeSetFixedStep()* on the SplittingStep object to specify the overall time step size.

## 8. Set optional inputs

## 9. Specify rootfinding problem

## 10. Advance solution in time

## 11. Get optional outputs

## 12. Deallocate memory for solution vector

## 13. Free solver memory

- If an ARKODE stepper module was used as a partition IVP integrator, call *SUNStepper\_Destroy()* and *ARKodeFree()* to free the memory allocated for that integrator.
- If a user-defined partition integrator was supplied, free the integrator content and call *SUNStepper\_Destroy()* to free the *SUNStepper* object.
- Call *ARKodeFree()* to free the memory allocated for the SplittingStep integration object.

## 14. Free the SUNContext object

## 15. Finalize MPI, if used

## 5.12.2 SplittingStep User-callable functions

This section describes the SplittingStep-specific functions that may be called by the user to setup and then solve an IVP using the SplittingStep time-stepping module.

As discussed in the main *ARKODE user-callable function introduction*, each of ARKODE's time-stepping modules clarifies the categories of user-callable functions that it supports. SplittingStep does not support any of the categories beyond the functions that apply for all time-stepping modules.

### 5.12.2.1 SplittingStep initialization functions

void \***SplittingStepCreate**(*SUNStepper* \*steppers, int partitions, *sunrealtype* t0, *N\_Vector* y0, *SUNContext* sunctx)

This function allocates and initializes memory for a problem to be solved using the SplittingStep time-stepping module in ARKODE.

#### Parameters

- **steppers** – an array of *SUNStepper* objects with one for each partition of the IVP. At minimum, they must implement the *SUNStepper\_Evolve()*, *SUNStepper\_Reset()*, *SUNStepper\_SetStopTime()*, and *SUNStepper\_SetStepDirection()* operations.
- **partitions** – the number of partitions,  $P > 1$ , in the IVP.
- **t0** – the initial value of  $t$ .
- **y0** – the initial condition vector  $y(t_0)$ .
- **sunctx** – the *SUNContext* object (see §4.2)

**Returns**

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

**Example usage:**

```
/* ARKODE objects for integrating individual partitions */
void *partition_mem[] = {NULL, NULL};

/* SUNSteppers to wrap the ARKODE objects */
SUNStepper steppers[] = {NULL, NULL};

/* create ARKODE objects, setting right-hand side functions and the
   initial condition */
partition_mem[0] = ERKStepCreate(f1, t0, y0, sunctx);
partition_mem[1] = ARKStepCreate(fe2, fi2, t0, y0, sunctx);

/* setup ARKODE objects */
. . .

/* create SUNStepper wrappers for the ARKODE memory blocks */
flag = ARKodeCreateSUNStepper(partition_mem[0], &stepper[0]);
flag = ARKodeCreateSUNStepper(partition_mem[1], &stepper[1]);

/* create a SplittingStep object with two partitions */
arkode_mem = SplittingStepCreate(steppers, 2, t0, y0, sunctx);
```

**Example codes:**

- `examples/arkode/C_serial/ark_advection_diffusion_reaction_splitting.c`
- `examples/arkode/C_serial/ark_analytic_partitioned.c`

Added in version 6.2.0.

**5.12.2.2 Optional inputs for IVP method selection**

int **SplittingStepSetCoefficients**(void \*arkode\_mem, *SplittingStepCoefficients* coefficients)

Specifies a set of coefficients for the operator splitting method.

**Parameters**

- **arkode\_mem** – pointer to the `SplittingStep` memory block.
- **coefficients** – the splitting coefficients for the method.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the `SplittingStep` memory is `NULL`
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

**Note**

For a description of the *SplittingStepCoefficients* type and related functions for creating splitting coefficients see §5.12.3.

This routine will be called by *ARKodeSetOptions()* when using the key “arkid.splitting\_coefficients\_name”, where *coefficients* is itself constructed by passing the command-line option to *SplittingStepCoefficients\_LoadCoefficientsByName()*.

**Warning**

This should not be used with *ARKodeSetOrder()*.

Added in version 6.2.0.

**5.12.2.3 Optional output functions**

int **SplittingStepGetNumEvolves**(void \*arkode\_mem, int partition, long int \*evolves)

Returns the number of times the *SUNStepper* for the given partition index has been evolved (so far).

**Parameters**

- **arkode\_mem** – pointer to the *SplittingStep* memory block.
- **partition** – index of the partition between 0 and  $P - 1$  or a negative number to indicate the total number across all partitions.
- **evolves** – number of *SUNStepper* evolves.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the *SplittingStep* memory was NULL
- **ARK\_ILL\_INPUT** – if *partition* was out of bounds

Added in version 6.2.0.

**5.12.2.4 SplittingStep re-initialization function**

To reinitialize the *SplittingStep* module for the solution of a new problem, where a prior call to *SplittingStepCreate()* has been made, the user must call the function *SplittingStepReInit()* and re-initialize each *SUNStepper*. The new problem must have the same size as the previous one. This routine retains the current settings for all *SplittingStep* module options and performs the same input checking and initializations that are done in *SplittingStepCreate()*, but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration, and deletes any previously-set *tstop* value specified via a call to *ARKodeSetStopTime()*. Following a successful call to *SplittingStepReInit()*, call *ARKodeEvolve()* again for the solution of the new problem.

One important use of the *SplittingStepReInit()* function is in the treating of jump discontinuities in the RHS function. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to this routine. To stop when the location of the discontinuity is known, simply make that location a value of *tout*. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS function *not* incorporate

the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS function (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

Another use of `SplittingStepReInit()` is changing the partitioning of the ODE and the `SUNStepper` objects used to evolve each partition.

int **SplittingStepReInit**(void \*arkode\_mem, `SUNStepper` \*steppers, int partitions, `sunrealtype` t0, `N_Vector` y0)

Provides required problem specifications and re-initializes the SplittingStep time-stepper module.

#### Parameters

- **arkode\_mem** – pointer to the SplittingStep memory block.
- **steppers** – an array of `SUNStepper` objects with one for each partition of the IVP. At minimum, they must implement the `SUNStepper_Evolve()`, `SUNStepper_Reset()`, `SUNStepper_SetStopTime()`, and `SUNStepper_SetStepDirection()` operations.
- **partitions** – the number of partitions,  $P > 1$ , in the IVP.
- **t0** – the initial value of  $t$ .
- **y0** – the initial condition vector  $y(t_0)$ .

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SplittingStep memory was NULL
- **ARK\_MEM\_FAIL** – if a memory allocation failed
- **ARK\_ILL\_INPUT** – if an argument has an illegal value

#### Warning

This function does not perform any re-initialization of the `SUNStepper` objects. It is up to the user to do this, if necessary.

#### Warning

If the number of partitions changes and coefficients were previously specified with `SplittingStepSetCoefficients()`, the coefficients will be reset since they are no longer compatible. Otherwise, all previously set options are retained but may be updated by calling the appropriate “Set” functions.

Added in version 6.2.0.

### 5.12.3 Operator Splitting Coefficients Data Structure

SplittingStep supplies several functions to construct operator splitting coefficients of various orders and partitions. There are also a number of built-in methods of fixed orders and partitions (see §5.12.3.2). Finally, a user may construct a custom set of coefficients and attach it with `SplittingStepSetCoefficients()`. The operator splitting coefficients are stored in a `SplittingStepCoefficients` object which is a pointer to a `SplittingStepCoefficientsMem` structure:

typedef *SplittingStepCoefficientsMem* \***SplittingStepCoefficients**

struct **SplittingStepCoefficientsMem**

Structure for storing the coefficients defining an operator splitting method.

As described in §2.8, an operator splitting method is defined by a vector  $\alpha \in \mathbb{R}^r$  and a tensor  $\beta \in \mathbb{R}^{r \times (s+1) \times P}$  where  $r$  is the number of sequential methods,  $s$  is the number of stages, and  $P$  is the number of partitions.

*sunrealtype* \***alpha**

An array containing the weight of each sequential method used to produce the overall operator splitting solution. The array is of length [sequential\_methods].

*sunrealtype* \*\*\***beta**

A three-dimensional array containing the time nodes of the partition integrations. The array has dimensions [sequential\_methods][stages+1][partitions].

int **sequential\_methods**

The number of sequential methods,  $r$ , combined to produce the overall operator splitting solution

int **stages**

The number of stages,  $s$

int **partitions**

The number of partitions,  $P$ , in the IVP

int **order**

The method order of accuracy

### 5.12.3.1 SplittingStepCoefficients Functions

This section describes the functions for creating and interacting with operator splitting coefficients. The function prototypes and as well as the relevant integer constants are defined `arkode/arkode_splittingstep.h`.

Table 5.9: SplittingStepCoefficients functions

Function name	Description
<i>SplittingStepCoefficients_LoadCoefficients()</i>	Load a pre-defined SplittingStepCoefficients by ID
<i>SplittingStepCoefficients_LoadCoefficientsByName()</i>	Load a pre-defined SplittingStepCoefficients by name
<i>SplittingStepCoefficients_IDToName()</i>	Convert a pre-defined SplittingStepCoefficients to its name
<i>SplittingStepCoefficients_LieTrotter()</i>	Create a Lie–Trotter splitting method
<i>SplittingStepCoefficients_Strang()</i>	Create a Strang splitting method
<i>SplittingStepCoefficients_SymmetricParallel()</i>	Create a symmetrization of the Lie–Trotter splitting method
<i>SplittingStepCoefficients_ThirdOrderSuzuki()</i>	Create a third order composition method of Suzuki
<i>SplittingStepCoefficients_TripleJump()</i>	Create an arbitrary order, three-jump composition method
<i>SplittingStepCoefficients_SuzukiFrac-tal()</i>	Create an arbitrary order, five-jump composition method
<i>SplittingStepCoefficients_Alloc()</i>	Allocate an empty <i>SplittingStepCoefficients</i> object
<i>SplittingStepCoefficients_Create()</i>	Create a new <i>SplittingStepCoefficients</i> object from coefficient arrays
<i>SplittingStepCoefficients_Copy()</i>	Create a copy of a <i>SplittingStepCoefficients</i> object
<i>SplittingStepCoefficients_Destroy()</i>	Deallocate a <i>SplittingStepCoefficients</i> object
<i>SplittingStepCoefficients_Write()</i>	Write the <i>SplittingStepCoefficients</i> object to an output file

*SplittingStepCoefficients* **SplittingStepCoefficients\_LoadCoefficients**(*ARKODE\_SplittingCoefficientsID* method)

Retrieves specified splitting coefficients. For further information on the current set of splitting coefficients and their corresponding identifiers, see §5.12.3.2.

#### Parameters

- **method** – the splitting coefficients identifier.

#### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if method was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_LoadCoefficientsByName**(const char \*method)

Retrieves specified splitting coefficients. For further information on the current set of splitting coefficients and their corresponding name, see §5.12.3.2.

#### Parameters

- **method** – the splitting coefficients identifier.

#### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if method was invalid or an allocation error occurred.

#### Note

This function is case sensitive.



Added in version 6.2.0.

const char \***SplittingStepCoefficients\_IDToName**(*ARKODE\_SplittingCoefficientsID* method)

Converts specified splitting coefficients ID to a string of the same name. For further information on the current set of splitting coefficients and their corresponding name, see §5.12.3.2.

#### Parameters

- **method** – the splitting coefficients identifier.

#### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if method was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_LieTrotter**(int partitions)

Create the coefficients for the first order Lie–Trotter splitting, see (2.20).

#### Parameters

- **partitions** – The number of partitions,  $P > 1$ , in the IVP.

#### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if partitions was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_Strang**(int partitions)

Create the coefficients for the second order Strang splitting [111], see (2.22).

#### Parameters

- **partitions** – The number of partitions,  $P > 1$ , in the IVP.

#### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if partitions was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_Parallel**(int partitions)

Create the coefficients for the first order parallel splitting method

$$y_n = \phi_{h_n}^1(y_{n-1}) + \phi_{h_n}^2(y_{n-1}) + \cdots + \phi_{h_n}^P(y_{n-1}) + (1 - P)y_{n-1}.$$

#### Parameters

- **partitions** – The number of partitions,  $P > 1$ , in the IVP.

#### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if partitions was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_SymmetricParallel**(int partitions)

Create the coefficients for the second order, symmetrized Lie–Trotter splitting [110]

$$y_n = \frac{1}{2} (L_{h_n}(y_{n-1}) + L_{h_n}^*(y_{n-1})),$$

where  $L_{h_n}$  is the Lie–Trotter splitting (2.20) and  $L_{h_n}^*$  is its adjoint (2.21).

### Parameters

- **partitions** – The number of partitions,  $P > 1$ , in the IVP.

### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if **partitions** was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_ThirdOrderSuzuki**(int partitions)

Create the coefficients for a splitting method of Suzuki [115]

$$y_n = (L_{p_1 h_n} \circ L_{p_2 h_n}^* \circ L_{p_3 h_n} \circ L_{p_4 h_n}^* \circ L_{p_5 h_n})(y_{n-1}),$$

where  $L_{h_n}$  is the Lie–Trotter splitting (2.20) and  $L_{h_n}^*$  is its adjoint (2.21). The parameters  $p_1, \dots, p_5$  are selected to give third order.

### Parameters

- **partitions** – The number of partitions,  $P > 1$ , in the IVP.

### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if **partitions** was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_TripleJump**(int partitions, int order)

Create the coefficients for the triple jump splitting method [31]

$$\begin{aligned} T_{h_n}^{[2]} &= S_{h_n}, \\ T_{h_n}^{[i+2]} &= T_{\gamma_1 h_n}^{[i]} \circ T_{(1-2\gamma_1)h_n}^{[i]} \circ T_{\gamma_1 h_n}^{[i]}, \\ y_n &= T_{h_n}^{[\text{order}]}(y_{n-1}), \end{aligned}$$

where  $S$  is the Strang splitting (2.22) and  $\gamma_1$  is selected to increase the order by two each recursion.

### Parameters

- **partitions** – The number of partitions,  $P > 1$ , in the IVP.
- **order** – A positive even number for the method order of accuracy.

### Returns

A *SplittingStepCoefficients* structure if successful or a NULL pointer if an argument was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_SuzukiFractal**(int partitions, int order)

Create the coefficients for the quintuple jump splitting method [114]

$$\begin{aligned} Q_{h_n}^{[2]} &= S_{h_n}, \\ Q_{h_n}^{[i+2]} &= Q_{\gamma_1 h_n}^{[i]} \circ Q_{\gamma_1 h_n}^{[i]} \circ Q_{(1-4\gamma_1)h_n}^{[i]} \circ Q_{\gamma_1 h_n}^{[i]} \circ Q_{\gamma_1 h_n}^{[i]}, \\ y_n &= Q_{h_n}^{[\text{order}]}(y_{n-1}), \end{aligned}$$

where  $S$  is the Strang splitting (2.22) and  $\gamma_1$  is selected to increase the order by two each recursion.

### Parameters

- **partitions** – The number of partitions,  $P > 1$ , in the IVP.

- **order** – A positive even number for the method order of accuracy.

**Returns**

A *SplittingStepCoefficients* structure if successful or a NULL pointer if an argument was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_Alloc**(int sequential\_methods, int stages, int partitions)

Allocates an empty *SplittingStepCoefficients* object.

**Parameters**

- **sequential\_methods** – The number of sequential methods,  $r \geq 1$ , combined to produce the overall operator splitting solution.
- **stages** – The number of stages,  $s \geq 1$ .
- **partitions** – The number of partitions,  $P > 1$ , in the IVP.

**Returns**

A *SplittingStepCoefficients* structure if successful or a NULL pointer if an argument was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_Create**(int sequential\_methods, int stages, int partitions, int order, *sunrealtype* \*alpha, *sunrealtype* \*beta)

Allocates a *SplittingStepCoefficients* object and fills it with the given values.

**Parameters**

- **sequential\_methods** – The number of sequential methods,  $r \geq 1$  combined to produce the overall operator splitting solution.
- **stages** – The number of stages,  $s \geq 1$ .
- **partitions** – The number of partitions,  $P > 1$  in the IVP.
- **order** – The method order of accuracy.
- **alpha** – An array of length sequential\_methods containing the weight of each sequential method used to produce the overall operator splitting solution.
- **beta** – An array of length sequential\_methods \* (stages+1) \* partitions containing the time nodes of the partition integrations in the C order

$$\begin{aligned} &\beta_{1,1,1}, \dots, \beta_{1,1,P}, \dots, \beta_{1,s+1,1}, \dots, \beta_{1,s+1,P}, \dots, \\ &\beta_{2,1,1}, \dots, \beta_{2,1,P}, \dots, \beta_{2,s+1,1}, \dots, \beta_{2,s+1,P}, \dots, \\ &\beta_{r,1,1}, \dots, \beta_{r,1,P}, \dots, \beta_{r,s+1,1}, \dots, \beta_{r,s+1,P}. \end{aligned}$$

**Returns**

A *SplittingStepCoefficients* structure if successful or a NULL pointer if an argument was invalid or an allocation error occurred.

Added in version 6.2.0.

*SplittingStepCoefficients* **SplittingStepCoefficients\_Copy**(*SplittingStepCoefficients* coefficients)

Creates copy of the given splitting coefficients.

**Parameters**

- **coefficients** – The splitting coefficients to copy.

**Returns**

A *SplittingStepCoefficients* structure if successful or a NULL pointer if an allocation error occurred.

Added in version 6.2.0.

void **SplittingStepCoefficients\_Destroy**(*SplittingStepCoefficients* \*coefficients)

Deallocate the splitting coefficients memory.

**Parameters**

- **coefficients** – A pointer to the splitting coefficients.

Added in version 6.2.0.

void **SplittingStepCoefficients\_Write**(*SplittingStepCoefficients* coefficients, FILE \*outfile)

Write the splitting coefficients to the provided file pointer.

**Parameters**

- **coefficients** – The splitting coefficients.
- **outfile** – Pointer to use for printing the splitting coefficients. It can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

Added in version 6.2.0.

### 5.12.3.2 Operator Splitting Coefficients

SplittingStep currently provides several pre-defined coefficients for problems with two partitions. We list the identifiers, order of accuracy, and relevant references for each in the table below. We use the naming convention

ARKODE\_SPLITTING\_<name>\_<stages>\_<order>\_<partitions>

Each of the splitting coefficients that are packaged with SplittingStep are specified by a unique ID having type:

enum **ARKODE\_SplittingCoefficientsID**

with values specified for each method below (e.g., `ARKODE_SPLITTING_LIE_TROTTER_1_1_2`).

Table 5.10: Operator splitting coefficients.

Table name	Order	Reference
ARKODE_SPLITTING_LIE_TROTTER_1_1_2	1	
ARKODE_SPLITTING_STRANG_2_2_2	2	[111]
ARKODE_SPLITTING_BEST_2_2_2	2	[14]
ARKODE_SPLITTING_SUZUKI_3_3_2	3	[115]
ARKODE_SPLITTING_RUTH_3_3_2	3	[90]
ARKODE_SPLITTING_YOSHIDA_4_4_2	4	[126]
ARKODE_SPLITTING_YOSHIDA_8_6_2	6	[126]

### 5.12.3.3 Default Operator Splitting Coefficients

The default SplittingStep coefficients are Lie–Trotter. If a particular order is requested with *ARKodeSetOrder()*, the following are the default for each order

Table 5.11: Default operator splitting coefficients by order.

Order	Default operator splitting coefficients
1	<a href="#"><i>SplittingStepCoefficients_LieTrotter()</i></a>
2	<a href="#"><i>SplittingStepCoefficients_Strang()</i></a>
3	<a href="#"><i>SplittingStepCoefficients_ThirdOrderSuzuki()</i></a>
4, 6, 8, ...	<a href="#"><i>SplittingStepCoefficients_TripleJump()</i></a>
5, 7, 9, ...	Warning: this will select a triple jump method of the next even order

## 5.13 Using the SPRKStep time-stepping module

This section is concerned with the use of the SPRKStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. Usage of SPRKStep follows that of the rest of ARKODE, and so in this section we primarily focus on those usage aspects that are specific to SPRKStep.

We note that of the ARKODE example programs located in the source code `examples/arkode` folder, the following demonstrate SPRKStep usage:

- `examples/arkode/C_serial/ark_harmonic_symplectic.c`
- `examples/arkode/C_serial/ark_damped_harmonic_symplectic.c`, and
- `examples/arkode/C_serial/ark_kepler.c`

### 5.13.1 SPRKStep User-callable functions

This section describes the SPRKStep-specific functions that may be called by the user to setup and then solve an IVP using the SPRKStep time-stepping module. The large majority of these routines merely wrap *underlying ARKODE functions*, and are now deprecated – each of these are clearly marked. However, some of these user-callable functions are specific to SPRKStep, as explained below.

As discussed in the main [\*ARKODE user-callable function introduction\*](#), each of ARKODE’s time-stepping modules clarifies the categories of user-callable functions that it supports. SPRKStep supports only the basic set of user-callable functions, and does not support any of the restricted groups (time adaptivity, implicit solvers, etc.).

SPRKStep does not have forcing function support when converted to a [\*SUNStepper\*](#) or [\*MRISStepInnerStepper\*](#). See [\*ARKodeCreateSUNStepper\(\)\*](#) and [\*ARKStepCreateMRISStepInnerStepper\(\)\*](#) for additional details.

#### 5.13.1.1 SPRKStep initialization and deallocation functions

void **\*SPRKStepCreate**([\*ARKRhsFn\*](#) f1, [\*ARKRhsFn\*](#) f2, [\*sunrealtype\*](#) t0, [\*N\\_Vector\*](#) y0, [\*SUNContext\*](#) sunctx)

This function allocates and initializes memory for a problem to be solved using the SPRKStep time-stepping module in ARKODE.

##### Parameters

- **f1** – the name of the C function (of type [\*ARKRhsFn\(\)\*](#)) defining  $f_1(t, q) = -\frac{\partial V(t, q)}{\partial q}$
- **f2** – the name of the C function (of type [\*ARKRhsFn\(\)\*](#)) defining  $f_2(t, p) = \frac{\partial T(t, p)}{\partial p}$
- **t0** – the initial value of  $t$
- **y0** – the initial condition vector  $y(t_0)$

- **sunctx** – the *SUNContext* object (see §4.2)

**Returns**

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

**Warning**

`SPRKStep` requires a partitioned problem where `f1` should only modify the `q` variables and `f2` should only modify the `p` variables (or vice versa). However, the vector passed to these functions is the full vector with both `p` and `q`. The ordering of the variables is determined implicitly by the user when they set the initial conditions.

void **SPRKStepFree**(void \*\*arkode\_mem)

This function frees the problem memory *arkode\_mem* created by *SPRKStepCreate()*.

**Parameters**

- **arkode\_mem** – pointer to the `SPRKStep` memory block.

Deprecated since version 6.1.0: Use *ARKodeFree()* instead.

### 5.13.1.2 Rootfinding initialization function

int **SPRKStepRootInit**(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 *SPRKStepCreate()*, and before *SPRKStepEvolve()*.

To disable the rootfinding feature after it has already been initialized, or to free memory associated with `SPRKStep`'s rootfinding module, call *SPRKStepRootInit()* with *nrtfn* = 0.

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

**Parameters**

- **arkode\_mem** – pointer to the `SPRKStep` memory block.
- **nrtfn** – number of functions  $g_i$ , an integer  $\geq 0$ .
- **g** – name of user-supplied function, of type *ARKRootFn*(), defining the functions  $g_i$  whose roots are sought.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the `SPRKStep` 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`.

Deprecated since version 6.1.0: Use *ARKodeRootInit()* instead.

### 5.13.1.3 SPRKStep solver function

int **SPRKStepEvolve**(void \*arkode\_mem, *sunrealtype* tout, *N\_Vector* yout, *sunrealtype* \*tret, int itask)

Integrates the ODE over an interval in  $t$ .

#### Parameters

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

The *ARK\_NORMAL* option causes the solver to take internal steps until it has just overtaken a user-specified output time, *tout*, in the direction of integration, i.e.  $t_{n-1} < tout \leq t_n$  for forward integration, or  $t_n \leq tout < t_{n-1}$  for backward integration. It will then compute an approximation to the solution  $y(tout)$  by interpolation (using one of the dense output routines described in §2.2).

The *ARK\_ONE\_STEP* option tells the solver to only take a single internal step,  $y_{n-1} \rightarrow y_n$ , and return the solution at that point,  $y_n$ , in the vector *yout*.

#### Return values

- **ARK\_SUCCESS** – if successful.
- **ARK\_ROOT\_RETURN** – if *SPRKStepEvolve()* succeeded, and found one or more roots. If the number of root functions, *nrtfn*, is greater than 1, call *SPRKStepGetRootInfo()* to see which  $g_i$  were found to have a root at (*\*tret*).
- **ARK\_TSTOP\_RETURN** – if *SPRKStepEvolve()* 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 *SPRKStepEvolve()* 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 are 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\_ERR\_FAILURE** – if error test failures occurred either too many times (*ark\_maxnef*) during one internal time step or occurred with  $|h| = h_{min}$ .
- **ARK\_VECTOROP\_ERR** – a vector operation error occurred.

#### Note

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

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 *SPRKStepEvolve()* failures.

Since interpolation may reduce the accuracy in the reported solution, if full method accuracy is desired the user should issue a call to `SPRKStepSetStopTime()` before the call to `SPRKStepEvolve()` to specify a fixed stop time to end the time step and return to the user. Upon return from `SPRKStepEvolve()`, a copy of the internal solution  $y_n$  will be returned in the vector *yout*. Once the integrator returns at a *tstop* time, any future testing for *tstop* is disabled (and can be re-enabled only through a new call to `SPRKStepSetStopTime()`). Interpolated outputs may or may not conserve the Hamiltonian. Our testing has shown that Lagrange interpolation typically performs well in this regard, while Hermite interpolation does not. As such, SPRKStep uses the Lagrange interpolation module by default.

On any error return in which one or more internal steps were taken by `SPRKStepEvolve()`, 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.

Deprecated since version 6.1.0: Use `ARKodeEvolve()` instead.

#### 5.13.1.4 Optional input functions

##### Optional inputs for SPRKStep

int **SPRKStepSetDefaults**(void \*arkode\_mem)

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

###### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.

###### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

###### Note

Does not change problem-defining function pointer *f* or the *user\_data* pointer.

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

Deprecated since version 6.1.0: Use `ARKodeSetDefaults()` instead.

int **SPRKStepSetInterpolantType**(void \*arkode\_mem, int itype)

Deprecated since version 6.1.0: This function is now a wrapper to `ARKodeSetInterpolantType()`, see the documentation for that function instead.

int **SPRKStepSetInterpolantDegree**(void \*arkode\_mem, int degree)

Specifies the degree of the polynomial interpolant used for dense output (i.e. interpolation of solution output values). Allowed values are between 0 and 5.

###### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **degree** – requested polynomial degree.



**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory or interpolation module are NULL
- **ARK\_INTERP\_FAIL** – if this is called after *SPRKStepEvolve()*
- **ARK\_ILL\_INPUT** – if an argument had an illegal value or the interpolation module has already been initialized

**Note**

This routine should be called *after* *SPRKStepCreate()* and *before* *SPRKStepEvolve()*. After the first call to *SPRKStepEvolve()* the interpolation degree may not be changed without first calling *SPRKStepReInit()*.

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

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

When  $q = 1$ , a linear interpolant is the default to ensure values obtained by the integrator are returned at the ends of the time interval.

Deprecated since version 6.1.0: Use *ARKodeSetInterpolantDegree()* instead.

int **SPRKStepSetFixedStep**(void \*arkode\_mem, *sunrealtype* hfixed)

Sets the time step size used within SPRKStep.

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **hfixed** – value of the fixed step size to use.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

Deprecated since version 6.1.0: Use *ARKodeSetFixedStep()* instead.

int **SPRKStepSetMaxNumSteps**(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 SPRKStep will return with an error.

Passing *mxsteps* = 0 results in SPRKStep using the default value (500).

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

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **mxsteps** – maximum allowed number of internal steps.

**Return values**

- **ARK\_SUCCESS** – if successful

- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

Deprecated since version 6.1.0: Use [ARKodeSetMaxNumSteps\(\)](#) instead.

int **SPRKStepSetStopTime**(void \*arkode\_mem, *sunrealtype* tstop)

Specifies the value of the independent variable  $t$  past which the solution is not to proceed.

The default is that no stop time is imposed.

Once the integrator returns at a stop time, any future testing for **tstop** is disabled (and can be re-enabled only through a new call to [SPRKStepSetStopTime\(\)](#)).

A stop time not reached before a call to [SPRKStepReInit\(\)](#) or [SPRKStepReset\(\)](#) will remain active but can be disabled by calling [SPRKStepClearStopTime\(\)](#).

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **tstop** – stopping time for the integrator.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

Deprecated since version 6.1.0: Use [ARKodeSetStopTime\(\)](#) instead.

int **SPRKStepClearStopTime**(void \*arkode\_mem)

Disables the stop time set with [SPRKStepSetStopTime\(\)](#).

The stop time can be re-enabled through a new call to [SPRKStepSetStopTime\(\)](#).

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL

Deprecated since version 6.1.0: Use [ARKodeClearStopTime\(\)](#) instead.

int **SPRKStepSetUserData**(void \*arkode\_mem, void \*user\_data)

Specifies the user data block *user\_data* and attaches it to the main SPRKStep memory block.

If specified, the pointer to *user\_data* is passed to all user-supplied functions for which it is an argument; otherwise NULL is passed.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **user\_data** – pointer to the user data.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

Deprecated since version 6.1.0: Use [ARKodeSetUserData\(\)](#) instead.

### Optional inputs for IVP method selection

Table 5.12: Optional inputs for IVP method selection

Optional input	Function name	Default
Set integrator method order	<a href="#">SPRKStepSetOrder()</a>	4
Set SPRK method	<a href="#">SPRKStepSetMethod()</a>	ARKODE_SPRK_MCLACHLAN_4_4
Set SPRK method by name	<a href="#">SPRKStepSetMethodName()</a>	“ARKODE_SPRK_MCLACHLAN_4_4”
Use compensated summation	<a href="#">SPRKStepSetUseCompensatedSums()</a>	false

int **SPRKStepSetOrder**(void \*arkode\_mem, int ord)

Specifies the order of accuracy for the SPRK integration method.

The allowed values are 1, 2, 3, 4, 5, 6, 8, 10. Any illegal input will result in the default value of 4.

Since *ord* affects the memory requirements for the internal SPRKStep memory block, it cannot be changed after the first call to [SPRKStepEvolve\(\)](#), unless [SPRKStepReInit\(\)](#) is called.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **ord** – requested order of accuracy.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

#### Warning

This overrides any previously set method so it should not be used with [SPRKStepSetMethod\(\)](#) or [SPRKStepSetMethodName\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeSetOrder\(\)](#) instead.

int **SPRKStepSetMethod**(void \*arkode\_mem, [ARKodeSPRKTable](#) sprk\_table)

Specifies the SPRK method.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **sprk\_table** – the SPRK method table.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL

- **ARK\_ILL\_INPUT** – if an argument had an illegal value

**Note**

No error checking is performed on the coefficients contained in the table to ensure its declared order of accuracy.

**Warning**

This should not be used with [ARKodeSetOrder\(\)](#).

int **SPRKStepSetMethodName**(void \*arkode\_mem, const char \*method)

Specifies the SPRK method by its name.

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **method** – the SPRK method name.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

**Note**

This routine will be called by [ARKodeSetOptions\(\)](#) when using the key “arkid.method\_name”.

**Warning**

This should not be used with [ARKodeSetOrder\(\)](#).

int **SPRKStepSetUseCompensatedSums**(void \*arkode\_mem, *sunbooleantype* onoff)

Specifies if *compensated summation (and the incremental form)* should be used where applicable.

This increases the computational cost by 2 extra vector operations per stage and an additional 5 per time step. It also requires one extra vector to be stored. However, it is significantly more robust to roundoff error accumulation.

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **onoff** – should compensated summation be used (1) or not (0)

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

Deprecated since version 6.4.0: Use [ARKodeSetUseCompensatedSums\(\)](#) instead.

### Rootfinding optional input functions

int **SPRKStepSetRootDirection**(void \*arkode\_mem, int \*rootdir)

Specifies the direction of zero-crossings to be located and returned.

The default behavior is to monitor for both zero-crossing directions.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **rootdir** – state array of length *nrtfn*, the number of root functions  $g_i$  (the value of *nrtfn* was supplied in the call to [SPRKStepRootInit\(\)](#)). 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 values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL
- **ARK\_ILL\_INPUT** – if an argument had an illegal value

Deprecated since version 6.1.0: Use [ARKodeSetRootDirection\(\)](#) instead.

int **SPRKStepSetNoInactiveRootWarn**(void \*arkode\_mem)

Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

SPRKStep 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), SPRKStep will issue a warning which can be disabled with this optional input function.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory is NULL

Deprecated since version 6.1.0: Use [ARKodeSetNoInactiveRootWarn\(\)](#) instead.

### 5.13.1.5 Interpolated output function

int **SPRKStepGetDky**(void \*arkode\_mem, *sunrealtype* t, int k, *N\_Vector* dky)

Computes the  $k$ -th derivative of the function  $y$  at the time  $t$ , i.e.,  $y^{(k)}(t)$ , 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. A user may access the values  $t_n$  and  $h_n$  via the functions [SPRKStepGetCurrentTime\(\)](#) and [SPRKStepGetLastStep\(\)](#), respectively.

This routine uses an interpolating polynomial of degree  $\min(\text{degree}, 5)$ , where *degree* is the argument provided to [SPRKStepSetInterpolantDegree\(\)](#). The user may request  $k$  in the range  $\{0, \dots, \min(\text{degree}, kmax)\}$  where *kmax* depends on the choice of interpolation module. For Hermite interpolants  $kmax = 5$  and for Lagrange interpolants  $kmax = 3$ .

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep 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 values**

- **ARK\_SUCCESS** – if successful
- **ARK\_BAD\_K** – if  $k$  is not in the range  $\{0, \dots, \min(\text{degree}, k_{\max})\}$ .
- **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 SPRKStep memory is NULL

**Note**

Dense outputs may or may not conserve the Hamiltonian. Our testing has shown that Lagrange interpolation typically performs well in this regard, while Hermite interpolation does not.

**Warning**

It is only legal to call this function after a successful return from [SPRKStepEvolve\(\)](#).

Deprecated since version 6.1.0: Use [ARKodeGetDky\(\)](#) instead.

### 5.13.1.6 Optional output functions

**Main solver optional output functions**

int **SPRKStepGetNumSteps**(void \*arkode\_mem, long int \*nsteps)

Returns the cumulative number of internal steps taken by the solver (so far).

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **nsteps** – number of steps taken in the solver.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumSteps\(\)](#) instead.

int **SPRKStepGetLastStep**(void \*arkode\_mem, *sunrealtype* \*hlast)

Returns the integration step size taken on the last successful internal step.

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **hlast** – step size taken on the last internal step.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetLastStep\(\)](#) instead.

int **SPRKStepGetCurrentStep**(void \*arkode\_mem, *sunrealtype* \*hcur)

Returns the integration step size to be attempted on the next internal step.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **hcur** – step size to be attempted on the next internal step.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentStep\(\)](#) instead.

int **SPRKStepGetCurrentTime**(void \*arkode\_mem, *sunrealtype* \*tcur)

Returns the current internal time reached by the solver.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **tcur** – current internal time reached.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetCurrentTime\(\)](#) instead.

int **SPRKStepGetCurrentState**(void \*arkode\_mem, *N\_Vector* \*ycur)

Returns the current internal solution reached by the solver.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **ycur** – current internal solution

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

#### Warning

Users should exercise extreme caution when using this function, as altering values of *ycur* may lead to undesirable behavior, depending on the particular use case and on when this routine is called.

Deprecated since version 6.1.0: Use [ARKodeGetCurrentState\(\)](#) instead.

```
int SPRKStepGetStepStats(void *arkode_mem, long int *nsteps, sunrealtype *hinused, sunrealtype *hlast,
                        sunrealtype *hcur, sunrealtype *tcur)
```

Returns many of the most useful optional outputs in a single call.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **nsteps** – number of steps taken in the solver.
- **hinused** – actual value of initial step size.
- **hlast** – step size taken on the last internal step.
- **hcur** – step size to be attempted on the next internal step.
- **tcur** – current internal time reached.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetStepStats\(\)](#) instead.

```
int SPRKStepPrintAllStats(void *arkode_mem, FILE *outfile, SUNOutputFormat fmt)
```

Outputs all of the integrator statistics.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **outfile** – pointer to output file.
- **fmt** – the output format:
  - [SUN\\_OUTPUTFORMAT\\_TABLE](#) – prints a table of values
  - [SUN\\_OUTPUTFORMAT\\_CSV](#) – prints a comma-separated list of key and value pairs e.g.,  
key1,value1,key2,value2,...

#### Return values

- **ARK\_SUCCESS** – if the output was successfully.
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL.
- **ARK\_ILL\_INPUT** – if an invalid formatting option was provided.

#### Note

The Python module `tools/suntools` provides utilities to read and output the data from a SUNDIALS CSV output file using the key and value pair format.

Deprecated since version 6.1.0: Use [ARKodePrintAllStats\(\)](#) instead.

```
char *SPRKStepGetReturnFlagName(long int flag)
```

Returns the name of the SPRKStep constant corresponding to *flag*. See [ARKODE Constants](#).

#### Parameters

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



**Returns**

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

**Warning**

The user is responsible for freeing the returned string.

Deprecated since version 6.1.0: Use [ARKodeGetReturnFlagName\(\)](#) instead.

int **SPRKStepGetNumStepAttempts**(void \*arkode\_mem, long int \*step\_attempts)

Returns the cumulative number of steps attempted by the solver (so far).

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **step\_attempts** – number of steps attempted by solver.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumStepAttempts\(\)](#) instead.

int **SPRKStepGetNumRhsEvals**(void \*arkode\_mem, long int \*nf1, long int \*nf2)

Returns the number of calls to the user's right-hand side functions,  $f_1$  and  $f_2$  (so far).

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **nf1** – number of calls to the user's  $f_1(t, p)$  function.
- **nf2** – number of calls to the user's  $f_2(t, q)$  function.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.2.0: Use [ARKodeGetNumRhsEvals\(\)](#) instead.

int **SPRKStepGetCurrentMethod**(void \*arkode\_mem, [ARKodeSPRKTable](#) \*sprk\_table)

Returns the SPRK method coefficient table currently in use by the solver.

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **sprk\_table** – pointer to the SPRK method table.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

int **SPRKStepGetUserData**(void \*arkode\_mem, void \*\*user\_data)

Returns the user data pointer previously set with [SPRKStepSetUserData\(\)](#).

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **user\_data** – memory reference to a user data pointer

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetUserData\(\)](#) instead.

**Rootfinding optional output functions**

int **SPRKStepGetRootInfo**(void \*arkode\_mem, int \*rootsfound)

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

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  $g_i$ .

The user must allocate space for *rootsfound* prior to calling this function.

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **rootsfound** – array of length *nrtfn* with the indices of the user functions  $g_i$  found to have a root (the value of *nrtfn* was supplied in the call to [SPRKStepRootInit\(\)](#)). For  $i = 0 \dots nrtfn-1$ , `rootsfound[i]` is nonzero if  $g_i$  has a root, and 0 if not.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetRootInfo\(\)](#) instead.

int **SPRKStepGetNumGEvals**(void \*arkode\_mem, long int \*ngevals)

Returns the cumulative number of calls made to the user's root function  $g$ .

**Parameters**

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **ngevals** – number of calls made to  $g$  so far.

**Return values**

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeGetNumGEvals\(\)](#) instead.

**General usability functions**

int **SPRKStepWriteParameters**(void \*arkode\_mem, FILE \*fp)

Outputs all SPRKStep solver parameters to the provided file pointer.

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.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **fp** – pointer to use for printing the solver parameters.

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL

Deprecated since version 6.1.0: Use [ARKodeWriteParameters\(\)](#) instead.

#### 5.13.1.7 SPRKStep re-initialization function

To reinitialize the SPRKStep module for the solution of a new problem, where a prior call to [SPRKStepCreate\(\)](#) has been made, the user must call the function [SPRKStepReInit\(\)](#). The new problem must have the same size as the previous one. This routine retains the current settings for all SPRKStep module options and performs the same input checking and initializations that are done in [SPRKStepCreate\(\)](#), but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration, and deletes any previously-set *tstop* value specified via a call to [SPRKStepSetStopTime\(\)](#). Following a successful call to [SPRKStepReInit\(\)](#), call [SPRKStepEvolve\(\)](#) again for the solution of the new problem.

The use of [SPRKStepReInit\(\)](#) requires that the number of Runge–Kutta stages, denoted by *s*, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the method order *q* is left unchanged.

One potential use of the [SPRKStepReInit\(\)](#) function is in the treating of jump discontinuities in the RHS function [116]. In lieu of including if statements within the RHS function to handle discontinuities, it may be more computationally efficient to stop at each point of discontinuity (e.g., through use of *tstop* or the rootfinding feature) and restart the integrator with a readjusted ODE model, using a call to this routine. We note that for the solution to retain temporal accuracy, the RHS function should not incorporate the discontinuity.

int **SPRKStepReInit**(void \*arkode\_mem, [ARKRhsFn](#) f1, [ARKRhsFn](#) f2, *sunrealtype* t0, *N\_Vector* y0)

Provides required problem specifications and re-initializes the SPRKStep time-stepper module.

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, [SPRKStepReInit\(\)](#) also sends an error message to the error handler function.

#### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **f1** – the name of the C function (of type [ARKRhsFn\(\)](#)) defining  $f1(t, q) = \frac{\partial V(t, q)}{\partial q}$
- **f2** – the name of the C function (of type [ARKRhsFn\(\)](#)) defining  $f2(t, p) = \frac{\partial T(t, p)}{\partial p}$
- **t0** – the initial value of *t*.
- **y0** – the initial condition vector  $y(t_0)$ .

#### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL
- **ARK\_MEM\_FAIL** – if a memory allocation failed

- **ARK\_ILL\_INPUT** – if an argument had an illegal value.

#### 5.13.1.8 SPRKStep reset function

int **SPRKStepReset**(void \*arkode\_mem, *sunrealtype* tR, *N\_Vector* yR)

Resets the current SPRKStep time-stepper module state to the provided independent variable value and dependent variable vector.

All previously set options are retained but may be updated by calling the appropriate “Set” functions.

If an error occurred, *SPRKStepReset()* also sends an error message to the error handler function.

##### Parameters

- **arkode\_mem** – pointer to the SPRKStep memory block.
- **tR** – the value of the independent variable  $t$ .
- **yR** – the value of the dependent variable vector  $y(t_R)$ .

##### Return values

- **ARK\_SUCCESS** – if successful
- **ARK\_MEM\_NULL** – if the SPRKStep memory was NULL
- **ARK\_MEM\_FAIL** – if a memory allocation failed
- **ARK\_ILL\_INPUTL** – if an argument had an illegal value.

##### Note

By default the next call to *SPRKStepEvolve()* will use the step size computed by SPRKStep prior to calling *SPRKStepReset()*.

Deprecated since version 6.1.0: Use *ARKodeReset()* instead.

## 5.14 Adjoint Sensitivity Analysis

Added in version 6.3.0.

The previous sections discuss using ARKODE for the integration of forward ODE models. This section discusses how to use ARKODE for adjoint sensitivity analysis as introduced in §2.19. To use ARKStep for adjoint sensitivity analysis (ASA), users simply setup the forward integration as usual (following §5.2) with a few differences. Below we provide an updated version of the ARKODE usage in section §5.2 where steps that are unchanged are *italicized*. The example code `examples/arkode/C_serial/ark_lotka_volterra_asa.c` demonstrates these steps in detail.

1. *Initialize parallel or multi-threaded environment, if appropriate.*
2. *Create the SUNDIALS simulation context object*
3. *Set problem dimensions, etc.*
4. *Set vector of initial values*
5. *Create ARKODE object*

6. Specify a fixed time step size.

Currently the discrete ASA capability only allows a fixed time step size to be used. Call `ARKodeSetFixedStep()` to set the time step.

7. *Set optional inputs*

8. *Specify rootfinding problem*

9. Create a `SUNAdjointCheckpointScheme` object

Create the `SUNAdjointCheckpointScheme` object by calling `SUNAdjointCheckpointScheme_Create_*`. Available `SUNAdjointCheckpointScheme` implementations are found in section §15.3.

10. Attach the checkpoint scheme object to ARKODE

Call `ARKodeSetAdjointCheckpointScheme()`.

11. *Advance solution in time*

12. *Get optional outputs*

13. Create the sensitivities vector with the terminal condition

The sensitivities vector must be an instance of the `ManyVector N_Vector implementation`. You will have one subvector for the initial condition sensitivities and an additional subvector if you want sensitivities with respect to parameters. The vectors should contain the terminal conditions for the adjoint problem. The first subvector should contain  $dg(t_f, y(t_f), p)/dy(t_f)$  and the second subvector should contain  $dg(t_f, y(t_f), p)/dp$ . The subvectors can be any implementation of the `N_Vector class`.

For example, in a problem with 10 state variables and 4 parameters using serial computations, the `ManyVector` can be constructed as follows:

```
sunindextype num_equations = 10;
sunindextype num_params    = 4;
N_Vector sensu0             = N_VNew_Serial(num_equations, sunctx);
N_Vector sensp              = N_VNew_Serial(num_params, sunctx);
N_Vector sens[2]            = {sensu0, sensp};
N_Vector sf                 = N_VNew_ManyVector(2, sens, sunctx);
// Set the terminal condition for the adjoint system, which
// should be the the gradient of our cost function at tf.
dgdu(u, sensu0, params);
dgdps(u, sensp, params);
```

14. Create the `SUNAdjointStepper` object

Call `ERKStepCreateAdjointStepper()` or `ARKStepCreateAdjointStepper()`.

15. Set optional ASA input

Refer to §15.2 for options.

16. Advance the adjoint sensitivity analysis ODE

Call `SUNAdjointStepper_Evolve()` or `SUNAdjointStepper_OneStep()`.

17. Get optional ASA outputs

Refer to §15.2 for options.

18. Deallocate memory for ASA objects

Deallocate the sensitivities vector, `SUNAdjointStepper`, and `SUNAdjointCheckpointScheme` objects.

19. *Deallocate memory for solution vector*

## 20. Free solver memory

Call `SUNStepper_Destroy()` and `ARKodeFree()` to free the memory allocated for the `SUNStepper` and `ARKODE` integrator objects.

21. Free the `SUNContext` object

## 22. Finalize MPI, if used

### 5.14.1 User Callable Functions

This section describes user-callable functions for performing adjoint sensitivity analysis with methods with `ERKStep` and `ARKStep`.

int **ERKStepCreateAdjointStepper**(void \*arkode\_mem, *SUNAdjRhsFn* f, *sunrealtype* tf, *N\_Vector* sf, *SUNContext* sunctx, *SUNAdjointStepper* \*adj\_stepper\_ptr)

Creates a *SUNAdjointStepper* object compatible with the provided `ERKStep` instance for integrating the adjoint sensitivity system (2.68).

#### Parameters

- **arkode\_mem** – a pointer to the `ERKStep` memory block.
- **f** – the adjoint right hand side function which implements  $\Lambda = f_y^*(t, y, p)\lambda$  and, if sensitivities with respect to parameters should be computed,  $\nu = f_p^*(t, y, p)\lambda$ .
- **tf** – the terminal time for the adjoint sensitivity system.
- **sf** – the sensitivity vector holding the adjoint system terminal condition. This must be an *NVECTOR\_MANYVECTOR* instance. The first subvector must be  $g_y^*(t_f, y(t_f), p) \in \mathbb{R}^N$ . If sensitivities to parameters should be computed, then the second subvector must be  $g_p^*(t_f, y(t_f), p) \in \mathbb{R}^{N_s}$ , otherwise only one subvector should be provided.
- **sunctx** – the SUNDIALS simulation context object.
- **adj\_stepper\_ptr** – the newly created *SUNAdjointStepper* object.

#### Return values

- **ARK\_SUCCESS** – if successful.
- **ARK\_MEM\_FAIL** – if a memory allocation failed.
- **ARK\_ILL\_INPUT** – if an argument has an illegal value.

Added in version 6.3.0.

#### Note

Currently fixed time steps must be used. Furthermore, the explicit stability function, inequality constraints, and relaxation features are not yet compatible as they require adaptive time steps.

int **ARKStepCreateAdjointStepper**(void \*arkode\_mem, *SUNAdjRhsFn* fe, *SUNAdjRhsFn* fi, *sunrealtype* tf, *N\_Vector* sf, *SUNContext* sunctx, *SUNAdjointStepper* \*adj\_stepper\_ptr)

Creates a *SUNAdjointStepper* object compatible with the provided `ARKStep` instance for integrating the adjoint sensitivity system (2.68).

#### Parameters

- **arkode\_mem** – a pointer to the `ARKStep` memory block.

- **fe** – the adjoint right hand side function which implements  $\Lambda = f_y^{E,*}(t, y, p)\lambda$  and, if sensitivities with respect to parameters should be computed,  $\nu = f_p^*(t, y, p)\lambda$ .
- **fi** – not yet support, the user should pass NULL.
- **tf** – the terminal time for the adjoint sensitivity system.
- **sf** – the sensitivity vector holding the adjoint system terminal condition. This must be a *NVECTOR\_MANYVECTOR* instance. The first subvector must be  $g_y^*(t_f, y(t_f), p) \in \mathbb{R}^N$ . If sensitivities to parameters should be computed, then the second subvector must be  $g_p^*(t_f, y(t_f), p) \in \mathbb{R}^{N_s}$ , otherwise only one subvector should be provided.
- **sunctx** – the SUNDIALS simulation context object.
- **adj\_stepper\_ptr** – the newly created *SUNAdjointStepper* object.

#### Return values

- **ARK\_SUCCESS** – if successful.
- **ARK\_MEM\_FAIL** – if a memory allocation failed.
- **ARK\_ILL\_INPUT** – if an argument has an illegal value.

Added in version 6.3.0.

#### Note

Currently only explicit methods with identity mass matrices are supported for ASA, and fixed time steps must be used. Furthermore, the explicit stability function, inequality constraints, and relaxation features are not yet compatible as they require adaptive time steps.





## Chapter 6

# Butcher Table Data Structure

To store a Butcher table,  $B$ , defining a Runge–Kutta method ARKODE provides the [ARKodeButcherTable](#) type and several related utility routines. We use the following notation

$$B \equiv \begin{array}{c|c} c & A \\ \hline q & b \\ p & \tilde{b} \end{array} = \begin{array}{c|cccc} c_1 & a_{1,1} & \cdots & a_{1,s-1} & a_{1,s} \\ c_2 & a_{2,1} & \cdots & a_{2,s-1} & a_{2,s} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_s & a_{s,1} & \cdots & a_{s,s-1} & a_{s,s} \\ \hline q & b_1 & \cdots & b_{s-1} & b_s \\ p & \tilde{b}_1 & \cdots & \tilde{b}_{s-1} & \tilde{b}_s \end{array}.$$

An [ARKodeButcherTable](#) is a pointer to the [ARKodeButcherTableMem](#) structure:

```
typedef ARKodeButcherTableMem *ARKodeButcherTable
```

```
struct ARKodeButcherTableMem
```

Structure for storing a Butcher table

```
int q
```

The method order of accuracy

```
int p
```

The embedding order of accuracy, typically  $q = p + 1$

```
int stages
```

The number of stages in the method,  $s$

```
sunrealtype **A
```

The method coefficients  $A \in \mathbb{R}^s$

```
sunrealtype *c
```

The method abscissa  $c \in \mathbb{R}^s$

```
sunrealtype *b
```

The method coefficients  $b \in \mathbb{R}^s$

```
sunrealtype *d
```

The method embedding coefficients  $\tilde{b} \in \mathbb{R}^s$

## 6.1 ARKodeButcherTable functions

Table 6.1: ARKodeButcherTable functions

Function name	Description
<code>ARKodeButcherTable_LoadERK()</code>	Retrieve a given explicit Butcher table by its unique ID
<code>ARKodeButcherTable_LoadERKByName()</code>	Retrieve a given explicit Butcher table by its unique name
<code>ARKodeButcherTable_ERKIDToName()</code>	Convert an explicit Butcher table ID to its name
<code>ARKodeButcherTable_LoadDIRK()</code>	Retrieve a given implicit Butcher table by its unique ID
<code>ARKodeButcherTable_LoadDIRKByName()</code>	Retrieve a given implicit Butcher table by its unique name
<code>ARKodeButcherTable_DIRKIDToName()</code>	Convert an implicit Butcher table ID to its name
<code>ARKodeButcherTable_Alloc()</code>	Allocate an empty Butcher table
<code>ARKodeButcherTable_Create()</code>	Create a new Butcher table
<code>ARKodeButcherTable_Copy()</code>	Create a copy of a Butcher table
<code>ARKodeButcherTable_Space()</code>	Get the Butcher table real and integer workspace size
<code>ARKodeButcherTable_Free()</code>	Deallocate a Butcher table
<code>ARKodeButcherTable_Write()</code>	Write the Butcher table to an output file
<code>ARKodeButcherTable_IsStifflyAccurate()</code>	Determine if $A[\text{stages} - 1][i] == b[i]$
<code>ARKodeButcherTable_CheckOrder()</code>	Check the order of a Butcher table
<code>ARKodeButcherTable_CheckARKOrder()</code>	Check the order of an ARK pair of Butcher tables

*ARKodeButcherTable* **ARKodeButcherTable\_LoadERK**(*ARKODE\_ERKTableID* emethod)

Retrieves a specified explicit Butcher table. The prototype for this function, as well as the integer names for each provided method, are defined in the header file `arkode/arkode_butcher_erk.h`. For further information on these tables and their corresponding identifiers, see §19.

**Arguments:**

- *emethod* – integer input specifying the given Butcher table.

**Return value:**

- *ARKodeButcherTable* structure if successful.
- NULL pointer if *emethod* was invalid.

*ARKodeButcherTable* **ARKodeButcherTable\_LoadERKByName**(const char \*emethod)

Retrieves a specified explicit Butcher table. The prototype for this function, as well as the names for each provided method, are defined in the header file `arkode/arkode_butcher_erk.h`. For further information on these tables and their corresponding names, see §19.

**Arguments:**

- *emethod* – name of the Butcher table.

**Return value:**

- *ARKodeButcherTable* structure if successful.
- NULL pointer if *emethod* was invalid or "ARKODE\_ERK\_NONE".

**Notes:**

This function is case sensitive.

const char \***ARKodeButcherTable\_ERKIDToName**(*ARKODE\_ERKTableID* emethod)

Converts a specified explicit Butcher table ID to a string of the same name. The prototype for this function, as well as the integer names for each provided method, are defined in the header file `arkode/arkode_butcher_erk.h`. For further information on these tables and their corresponding identifiers, see §19.

**Arguments:**

- *emethod* – integer input specifying the given Butcher table.

**Return value:**

- The name associated with *emethod*.
- NULL pointer if *emethod* was invalid.

Added in version 6.1.0.

*ARKodeButcherTable* **ARKodeButcherTable\_LoadDIRK**(*ARKODE\_DIRKTableID* imethod)

Retrieves a specified diagonally-implicit Butcher table. The prototype for this function, as well as the integer names for each provided method, are defined in the header file `arkode/arkode_butcher_dirk.h`. For further information on these tables and their corresponding identifiers, see §19.

**Arguments:**

- *imethod* – integer input specifying the given Butcher table.

**Return value:**

- *ARKodeButcherTable* structure if successful.
- NULL pointer if *imethod* was invalid.

*ARKodeButcherTable* **ARKodeButcherTable\_LoadDIRKByName**(const char \*imethod)

Retrieves a specified diagonally-implicit Butcher table. The prototype for this function, as well as the names for each provided method, are defined in the header file `arkode/arkode_butcher_dirk.h`. For further information on these tables and their corresponding names, see §19.

**Arguments:**

- *imethod* – name of the Butcher table.

**Return value:**

- *ARKodeButcherTable* structure if successful.
- NULL pointer if *imethod* was invalid or "ARKODE\_DIRK\_NONE".

**Notes:**

This function is case sensitive.

const char \***ARKodeButcherTable\_DIRKIDToName**(*ARKODE\_DIRKTableID* imethod)

Converts a specified diagonally-implicit Butcher table ID to a string of the same name. The prototype for this function, as well as the integer names for each provided method, are defined in the header file `arkode/arkode_butcher_dirk.h`. For further information on these tables and their corresponding identifiers, see §19.

**Arguments:**

- *imethod* – integer input specifying the given Butcher table.

**Return value:**

- The name associated with *imethod*.
- NULL pointer if *imethod* was invalid.

Added in version 6.1.0.

*ARKodeButcherTable* **ARKodeButcherTable\_Alloc**(int stages, *sunbooleantype* embedded)

Allocates an empty Butcher table.

**Arguments:**

- *stages* – the number of stages in the Butcher table.
- *embedded* – flag denoting whether the Butcher table has an embedding (SUNTRUE) or not (SUNFALSE).

**Return value:**

- *ARKodeButcherTable* structure if successful.
- NULL pointer if *stages* was invalid or an allocation error occurred.

*ARKodeButcherTable* **ARKodeButcherTable\_Create**(int s, int q, int p, *sunrealtype* \*c, *sunrealtype* \*A, *sunrealtype* \*b, *sunrealtype* \*d)

Allocates a Butcher table and fills it with the given values.

**Arguments:**

- *s* – number of stages in the RK method.
- *q* – global order of accuracy for the RK method.
- *p* – global order of accuracy for the embedded RK method.
- *c* – array (of length *s*) of stage times for the RK method.
- *A* – array of coefficients defining the RK stages. This should be stored as a 1D array of size *s*\**s*, in row-major order.
- *b* – array of coefficients (of length *s*) defining the time step solution.
- *d* – array of coefficients (of length *s*) defining the embedded solution.

**Return value:**

- *ARKodeButcherTable* structure if successful.
- NULL pointer if *stages* was invalid or an allocation error occurred.

**Notes:**

If the method does not have an embedding then *d* should be NULL and *p* should be equal to zero.

**Warning**

When calling this function from Fortran, it is important to note that A is expected to be in row-major ordering.

*ARKodeButcherTable* **ARKodeButcherTable\_Copy**(*ARKodeButcherTable* B)

Creates copy of the given Butcher table.

**Arguments:**

- *B* – the Butcher table to copy.

**Return value:**

- *ARKodeButcherTable* structure if successful.
- NULL pointer an allocation error occurred.

void **ARKodeButcherTable\_Space**(*ARKodeButcherTable* B, *sunindextype* \*liw, *sunindextype* \*lrw)

Get the real and integer workspace size for a Butcher table.

**Arguments:**

- *B* – the Butcher table.

- *lenrw* – the number of `sunrealtype` values in the Butcher table workspace.
- *leniw* – the number of integer values in the Butcher table workspace.

**Return value:**

- `ARK_SUCCESS` if successful.
- `ARK_MEM_NULL` if the Butcher table memory was NULL.

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

void **ARKodeButcherTable\_Free**(*ARKodeButcherTable* B)

Deallocate the Butcher table memory.

**Arguments:**

- *B* – the Butcher table.

void **ARKodeButcherTable\_Write**(*ARKodeButcherTable* B, FILE \*outfile)

Write the Butcher table to the provided file pointer.

**Arguments:**

- *B* – the Butcher table.
- *outfile* – pointer to use for printing the Butcher table.

**Notes:**

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

*sunbooleantype* **ARKodeButcherTable\_IsStifflyAccurate**(*ARKodeButcherTable* B)

Determine if the table satisfies  $A[\text{stages} - 1][i] == b[i]$

**Arguments:**

- *B* – the Butcher table.

**Returns**

- `SUNTRUE` if the method is “stiffly accurate”, otherwise returns `SUNFALSE`

Added in version v5.7.0.

int **ARKodeButcherTable\_CheckOrder**(*ARKodeButcherTable* B, int \*q, int \*p, FILE \*outfile)

Determine the analytic order of accuracy for the specified Butcher table. The analytic (necessary) conditions are checked up to order 6. For orders greater than 6 the Butcher simplifying (sufficient) assumptions are used.

**Arguments:**

- *B* – the Butcher table.
- *q* – the measured order of accuracy for the method.
- *p* – the measured order of accuracy for the embedding; 0 if the method does not have an embedding.
- *outfile* – file pointer for printing results; NULL to suppress output.

**Return value:**

- 0 – success, the measured vales of *q* and *p* match the values of *q* and *p* in the provided Butcher tables.
- 1 – warning, the values of *q* and *p* in the provided Butcher tables are *lower* than the measured values, or the measured values achieve the *maximum order* possible with this function and the values of *q* and *p* in the provided Butcher tables table are higher.
- -1 – failure, the values of *q* and *p* in the provided Butcher tables are *higher* than the measured values.

- -2 – failure, the input Butcher table or critical table contents are NULL.

**Notes:**

For embedded methods, if the return flags for  $q$  and  $p$  would differ, failure takes precedence over warning, which takes precedence over success.

int **ARKodeButcherTable\_CheckARKOrder**(*ARKodeButcherTable* B1, *ARKodeButcherTable* B2, int \*q, int \*p, FILE \*outfile)

Determine the analytic order of accuracy (up to order 6) for a specified ARK pair of Butcher tables.

**Arguments:**

- $B1$  – a Butcher table in the ARK pair.
- $B2$  – a Butcher table in the ARK pair.
- $q$  – the measured order of accuracy for the method.
- $p$  – the measured order of accuracy for the embedding; 0 if the method does not have an embedding.
- *outfile* – file pointer for printing results; NULL to suppress output.

**Return value:**

- 0 – success, the measured values of  $q$  and  $p$  match the values of  $q$  and  $p$  in the provided Butcher tables.
- 1 – warning, the values of  $q$  and  $p$  in the provided Butcher tables are *lower* than the measured values, or the measured values achieve the *maximum order* possible with this function and the values of  $q$  and  $p$  in the provided Butcher tables are higher.
- -1 – failure, the input Butcher tables or critical table contents are NULL.

**Notes:**

For embedded methods, if the return flags for  $q$  and  $p$  would differ, warning takes precedence over success.

## Chapter 7

# SPRK Method Table Structure

To store the pair of Butcher tables defining a SPRK method of order  $q$  ARKODE provides the [ARKodeSPRKTable](#) type and several related utility routines. We use the following notation

$$B \equiv \frac{c}{b} \Big| \frac{A}{b} = \begin{array}{c|cccc} c_1 & 0 & \cdots & 0 & 0 \\ c_2 & a_1 & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ c_s & a_1 & \cdots & a_{s-1} & 0 \\ \hline & a_1 & \cdots & a_{s-1} & a_s \end{array} \quad \hat{B} \equiv \frac{\hat{c}}{\hat{b}} \Big| \frac{\hat{A}}{\hat{b}} = \begin{array}{c|cccc} \hat{c}_1 & \hat{a}_1 & \cdots & 0 & 0 \\ \hat{c}_2 & \hat{a}_1 & \hat{a}_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \hat{c}_s & \hat{a}_1 & \hat{a}_2 & \cdots & \hat{a}_s \\ \hline & \hat{a}_1 & \hat{a}_2 & \cdots & \hat{a}_s \end{array}$$

where  $B$  and  $\hat{B}$  contain the coefficients for the explicit and diagonally implicit tables, respectively. We use a compact storage of these coefficients in terms of two arrays, one for  $a$  values and one for  $\hat{a}$  values. The abscissae (only relevant for non-autonomous problems) are computed dynamically as  $c_j = \sum_{i=1}^j a_i$  and  $\hat{c}_j = \sum_{i=1}^j \hat{a}_i$ , respectively [36, 64]. The [ARKodeSPRKTable](#) type is a pointer to the [ARKodeSPRKTableMem](#) structure:

```
typedef ARKodeSPRKTableMem *ARKodeSPRKTable
```

```
type ARKodeSPRKTableMem
```

Structure representing the SPRK method that holds the method coefficients.

```
int q
```

The method order of accuracy.

```
int stages
```

The number of stages.

```
sunrealtype *a
```

Array of coefficients that generate the explicit Butcher table. `a[i]` is the coefficient appearing in column `i+1`.

```
sunrealtype *ahat
```

Array of coefficients that generate the diagonally-implicit Butcher table. `ahat[i]` is the coefficient appearing in column `i`.

## 7.1 ARKodeSPRKTable functions

Table 7.1: ARKodeSPRKTable functions

Function name	Description
<a href="#"><i>ARKodeSPRKTable_Alloc()</i></a>	Allocate an empty table
<a href="#"><i>ARKodeSPRKTable_Load()</i></a>	Load SPRK method using an identifier
<a href="#"><i>ARKodeSPRKTable_LoadByName()</i></a>	Load SPRK method using a string version of the identifier
<a href="#"><i>ARKodeSPRKTable_Create()</i></a>	Create a new table
<a href="#"><i>ARKodeSPRKTable_Copy()</i></a>	Create a copy of a table
<a href="#"><i>ARKodeSPRKTable_Space()</i></a>	Get the table real and integer workspace size
<a href="#"><i>ARKodeSPRKTable_Free()</i></a>	Deallocate a table

*ARKodeSPRKTable* **ARKodeSPRKTable\_Create**(int stages, int q, const *sunrealtype* \*a, const *sunrealtype* \*ahat)

Creates and allocates an [\*ARKodeSPRKTable\*](#) with the specified number of stages and the coefficients provided.

### Parameters

- **stages** – The number of stages.
- **q** – The order of the method.
- **a** – An array of the coefficients for the a table.
- **ahat** – An array of the coefficients for the ahat table.

### Returns

[\*ARKodeSPRKTable\*](#) for the loaded method.

*ARKodeSPRKTable* **ARKodeSPRKTable\_Alloc**(int stages)

Allocate memory for an [\*ARKodeSPRKTable\*](#) with the specified number of stages.

### Parameters

- **stages** – The number of stages.

### Returns

[\*ARKodeSPRKTable\*](#) for the loaded method.

*ARKodeSPRKTable* **ARKodeSPRKTable\_Load**(*ARKODE\_SPRKMethodID* id)

Load the [\*ARKodeSPRKTable\*](#) for the specified method ID.

### Parameters

- **id** – The ID of the SPRK method, see *Symplectic Partitioned Butcher tables*.

### Returns

[\*ARKodeSPRKTable\*](#) for the loaded method.

*ARKodeSPRKTable* **ARKodeSPRKTable\_LoadByName**(const char \*method)

Load the [\*ARKodeSPRKTable\*](#) for the specified method name.

### Parameters

- **method** – The name of the SPRK method, see *Symplectic Partitioned Butcher tables*.

### Returns

[\*ARKodeSPRKTable\*](#) for the loaded method.



*ARKodeSPRKTable* **ARKodeSPRKTable\_Copy**(*ARKodeSPRKTable* sprk\_table)

Create a copy of the *ARKodeSPRKTable*.

**Parameters**

- **sprk\_table** – The *ARKodeSPRKTable* to copy.

**Returns**

Pointer to the copied *ARKodeSPRKTable*.

void **ARKodeSPRKTable\_Write**(*ARKodeSPRKTable* sprk\_table, FILE \*outfile)

Write the *ARKodeSPRKTable* out to the file.

**Parameters**

- **sprk\_table** – The *ARKodeSPRKTable* to write.
- **outfile** – The FILE that will be written to.

void **ARKodeSPRKTable\_Space**(*ARKodeSPRKTable* sprk\_table, *sunindextype* \*liw, *sunindextype* \*lrw)

Get the workspace sizes required for the *ARKodeSPRKTable*.

**Parameters**

- **sprk\_table** – The *ARKodeSPRKTable*.
- **liw** – Pointer to store the integer workspace size.
- **lrw** – Pointer to store the real workspace size.

Deprecated since version 6.3.0: Work space functions will be removed in version 8.0.0.

void **ARKodeSPRKTable\_Free**(*ARKodeSPRKTable* sprk\_table)

Free the memory allocated for the *ARKodeSPRKTable*.

**Parameters**

- **sprk\_table** – The *ARKodeSPRKTable* to free.

int **ARKodeSPRKTable\_ToButcher**(*ARKodeSPRKTable* sprk\_table, *ARKodeButcherTable* \*a\_ptr,  
*ARKodeButcherTable* \*b\_ptr)

Convert the *ARKodeSPRKTable* to the Butcher table representation.

**Parameters**

- **sprk\_table** – The *ARKodeSPRKTable*.
- **a\_ptr** – Pointer to store the explicit Butcher table.
- **b\_ptr** – Pointer to store the diagonally-implicit Butcher table.



## Chapter 8

# 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.

### 8.1 Description of the NVECTOR Modules

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 vector implementation. Users can provide a custom vector implementation or use one provided with SUNDIALS. The generic operations are described below. In the sections following, the implementations provided with SUNDIALS are described.

An *N\_Vector* is a pointer to the *\_generic\_N\_Vector* structure:

```
typedef struct _generic_N_Vector *N_Vector
```

```
struct _generic_N_Vector
```

The structure defining the SUNDIALS vector class.

void \***content**

Pointer to vector-specific member data.

*N\_Vector\_Ops* **ops**

A virtual table of vector operations provided by a specific implementation.

*SUNContext* **sunctx**

The SUNDIALS simulation context

The virtual table structure is defined as

```
typedef _generic_N_Vector_Ops *N_Vector_Ops
```

**struct \_generic\_N\_Vector\_Ops**

The structure defining *N\_Vector* operations.

*N\_Vector\_ID* (\***nvgetvectorid**)(*N\_Vector*)

The function implementing *N\_VGetVectorID()*

*N\_Vector* (\***nvclone**)(*N\_Vector*)

The function implementing *N\_VClone()*

*N\_Vector* (\***nvcloneempty**)(*N\_Vector*)

The function implementing *N\_VCloneEmpty()*

void (\***nvdestroy**)(*N\_Vector*)

The function implementing *N\_VDestroy()*

void (\***nvspace**)(*N\_Vector*, *sunindextype\**, *sunindextype\**)

The function implementing *N\_VSpace()*

*sunrealtype* \*(\***nvgetarraypointer**)(*N\_Vector*)

The function implementing *N\_VGetArrayPointer()*

*sunrealtype* \*(\***nvgetdevicearraypointer**)(*N\_Vector*)

The function implementing *N\_VGetDeviceArrayPointer()*

void (\***nvsetarraypointer**)(*sunrealtype\**, *N\_Vector*)

The function implementing *N\_VSetArrayPointer()*

*SUNComm* (\***nvgetcommunicator**)(*N\_Vector*)

The function implementing *N\_VGetCommunicator()*

*sunindextype* (\***nvgetlength**)(*N\_Vector*)

The function implementing *N\_VGetLength()*

*sunindextype* (\***nvgetlocallength**)(*N\_Vector*)

The function implementing *N\_VGetLocalLength()*

void (\***nvlinearsum**)(*sunrealtype*, *N\_Vector*, *sunrealtype*, *N\_Vector*, *N\_Vector*)

The function implementing *N\_VLinearSum()*

void (\***nvconst**)(*sunrealtype*, *N\_Vector*)

The function implementing *N\_VConst()*

void (\***nvprod**)(*N\_Vector*, *N\_Vector*, *N\_Vector*)

The function implementing *N\_VProd()*

void (\***nvdiv**)(*N\_Vector*, *N\_Vector*, *N\_Vector*)

The function implementing *N\_VDiv()*

void (\***nvscale**)(*sunrealtype*, *N\_Vector*, *N\_Vector*)

The function implementing *N\_VScale()*

void (\***nvabs**)(*N\_Vector*, *N\_Vector*)

The function implementing *N\_VAbs()*

void (\***nvinv**)(*N\_Vector*, *N\_Vector*)

The function implementing *N\_VInv()*

`void (*nvaddconst)(N_Vector, sunrealtype, N_Vector)`  
 The function implementing `N_VAddConst()`

`sunrealtype (*nvdotprod)(N_Vector, N_Vector)`  
 The function implementing `N_VDotProd()`

`sunrealtype (*nvmaxnorm)(N_Vector)`  
 The function implementing `N_VMaxNorm()`

`sunrealtype (*nvwrmsnorm)(N_Vector, N_Vector)`  
 The function implementing `N_VWrmsNorm()`

`sunrealtype (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector)`  
 The function implementing `N_VWrmsNormMask()`

`sunrealtype (*nvmin)(N_Vector)`  
 The function implementing `N_VMin()`

`sunrealtype (*nvwl2norm)(N_Vector, N_Vector)`  
 The function implementing `N_VWL2Norm()`

`sunrealtype (*nv1lnorm)(N_Vector)`  
 The function implementing `N_VL1Norm()`

`void (*nvcompare)(sunrealtype, N_Vector, N_Vector)`  
 The function implementing `N_VCompare()`

`sunboolean_type (*nvinvtest)(N_Vector, N_Vector)`  
 The function implementing `N_VInvTest()`

`sunboolean_type (*nvconstrmask)(N_Vector, N_Vector, N_Vector)`  
 The function implementing `N_VConstrMask()`

`sunrealtype (*nvminquotient)(N_Vector, N_Vector)`  
 The function implementing `N_VMinQuotient()`

`SUNErrCode (*nvlinearcombination)(int, sunrealtype*, N_Vector*, N_Vector)`  
 The function implementing `N_VLinearCombination()`

`SUNErrCode (*nvscaleaddmulti)(int, sunrealtype*, N_Vector, N_Vector*, N_Vector*)`  
 The function implementing `N_VScaleAddMulti()`

`SUNErrCode (*nvdotprodmulti)(int, N_Vector, N_Vector*, sunrealtype*)`  
 The function implementing `N_VDotProdMulti()`

`SUNErrCode (*nvlinearsumvectorarray)(int, sunrealtype, N_Vector*, sunrealtype, N_Vector*, N_Vector*)`  
 The function implementing `N_VLinearSumVectorArray()`

`SUNErrCode (*nvscalevectorarray)(int, sunrealtype*, N_Vector*, N_Vector*)`  
 The function implementing `N_VScaleVectorArray()`

`SUNErrCode (*nvconstvectorarray)(int, sunrealtype, N_Vector*)`  
 The function implementing `N_VConstVectorArray()`

`SUNErrCode (*nvwrmsnormvectorarray)(int, N_Vector*, N_Vector*, sunrealtype*)`  
 The function implementing `N_VWrmsNormVectorArray()`

*SUNErrorCode* (\***nvwrmsnormmaskvectorarray**)(int, *N\_Vector*\*, *N\_Vector*\*, *N\_Vector*, *sunrealtype*\*)

The function implementing *N\_VWrmsNormMaskVectorArray()*

*SUNErrorCode* (\***nvscaleaddmultivectorarray**)(int, int, *sunrealtype*\*, *N\_Vector*\*, *N\_Vector*\*\*, *N\_Vector*\*\*)

The function implementing *N\_VScaleAddMultiVectorArray()*

*SUNErrorCode* (\***nvlinearcombinationvectorarray**)(int, int, *sunrealtype*\*, *N\_Vector*\*\*, *N\_Vector*\*)

The function implementing *N\_VLinearCombinationVectorArray()*

*sunrealtype* (\***nvdotprodlocal**)(*N\_Vector*, *N\_Vector*)

The function implementing *N\_VDotProdLocal()*

*sunrealtype* (\***nvmaxnormlocal**)(*N\_Vector*)

The function implementing *N\_VMaxNormLocal()*

*sunrealtype* (\***nvminlocal**)(*N\_Vector*)

The function implementing *N\_VMinLocal()*

*sunrealtype* (\***nv1lnormlocal**)(*N\_Vector*)

The function implementing *N\_VL1NormLocal()*

*sunboolean* (\***nvinvtestlocal**)(*N\_Vector*, *N\_Vector*)

The function implementing *N\_VInvTestLocal()*

*sunboolean* (\***nvconstrmasklocal**)(*N\_Vector*, *N\_Vector*, *N\_Vector*)

The function implementing *N\_VConstrMaskLocal()*

*sunrealtype* (\***nvminquotientlocal**)(*N\_Vector*, *N\_Vector*)

The function implementing *N\_VMinQuotientLocal()*

*sunrealtype* (\***nvwsqrsumlocal**)(*N\_Vector*, *N\_Vector*)

The function implementing *N\_VWSqrSumLocal()*

*sunrealtype* (\***nvwsqrsummasklocal**)(*N\_Vector*, *N\_Vector*, *N\_Vector*)

The function implementing *N\_VWSqrSumMaskLocal()*

*SUNErrorCode* (\***nvdotprodmultilocal**)(int, *N\_Vector*, *N\_Vector*\*, *sunrealtype*\*)

The function implementing *N\_VDotProdMultiLocal()*

*SUNErrorCode* (\***nvdotprodmultiallreduce**)(int, *N\_Vector*, *sunrealtype*\*)

The function implementing *N\_VDotProdMultiAllReduce()*

*SUNErrorCode* (\***nvbufsize**)(*N\_Vector*, *sunindex*\*)

The function implementing *N\_VBufSize()*

*SUNErrorCode* (\***nvbufpack**)(*N\_Vector*, void\*)

The function implementing *N\_VBufPack()*

*SUNErrorCode* (\***nvbufunpack**)(*N\_Vector*, void\*)

The function implementing *N\_VBufUnpack()*

void (\***nvprint**)(*N\_Vector*)

The function implementing *N\_VPrint()*

```
void (*nvprintf)(N_Vector, FILE*)
```

The function implementing `N_VPrintfFile()`

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 operation  $z \leftarrow cx$  for vectors  $x$  and  $z$  and a scalar  $c$ :

```
void N_VScale(sunrealtype c, N_Vector x, N_Vector z) {
    z->ops->nvscale(c, x, z);
}
```

§8.2 contains a complete list of all standard vector operations defined by the generic NVECTOR module. §8.2.2, §8.2.3, §8.2.4, §8.2.5, and §8.2.6 list *optional* fused, vector array, local reduction, single buffer reduction, and exchange operations, respectively.

Fused and vector array operations (see §8.2.2 and §8.2.3) are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines a fused or vector array operation as `NULL`, the generic NVECTOR module will automatically call standard vector operations as necessary to complete the desired operation. In all SUNDIALS-provided NVECTOR implementations, all fused and vector array operations are disabled by default. However, these implementations provide additional user-callable functions to enable/disable any or all of the fused and vector array operations. See the following sections for the implementation specific functions to enable/disable operations.

Local reduction operations (see §8.2.4) are similarly intended to reduce parallel communication on distributed memory systems, particularly when NVECTOR objects are combined together within an `NVECTOR_MANYVECTOR` object (see §8.17). If a particular NVECTOR implementation defines a local reduction operation as `NULL`, the `NVECTOR_MANYVECTOR` module will automatically call standard vector reduction operations as necessary to complete the desired operation. All SUNDIALS-provided NVECTOR implementations include these local reduction operations, which may be used as templates for user-defined implementations.

The single buffer reduction operations (§8.2.5) are used in low-synchronization methods to combine separate reductions into one `MPI_Allreduce` call.

The exchange operations (see §8.2.6) are intended only for use with the XBraid library for parallel-in-time integration (accessible from ARKODE) and are otherwise unused by SUNDIALS packages.

### 8.1.1 NVECTOR Utility Functions

The generic NVECTOR module also defines several utility functions to aid in creation and management of arrays of `N_Vector` objects – these functions are particularly useful for Fortran users to utilize the `NVECTOR_MANYVECTOR` or SUNDIALS’ sensitivity-enabled packages CVODES and IDAS.

The functions `N_VCloneVectorArray()` and `N_VCloneVectorArrayEmpty()` create (by cloning) an array of *count* variables of type `N_Vector`, each of the same type as an existing `N_Vector` input:

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w)
```

Clones an array of *count* `N_Vector` objects, allocating their data arrays (similar to `N_VClone()`).

#### Arguments:

- *count* – number of `N_Vector` objects to create.
- *w* – template `N_Vector` to clone.

#### Return value:

- pointer to a new `N_Vector` array on success.

- NULL pointer on failure.

*N\_Vector* \***N\_VCloneVectorArrayEmpty**(int count, *N\_Vector* w)

Clones an array of count *N\_Vector* objects, leaving their data arrays unallocated (similar to *N\_VCloneEmpty()*).

**Arguments:**

- count – number of *N\_Vector* objects to create.
- w – template *N\_Vector* to clone.

**Return value:**

- pointer to a new *N\_Vector* array on success.
- NULL pointer on failure.

An array of variables of type *N\_Vector* can be destroyed by calling *N\_VDestroyVectorArray()*:

void **N\_VDestroyVectorArray**(*N\_Vector* \*vs, int count)

Destroys an array of count *N\_Vector* objects.

**Arguments:**

- vs – *N\_Vector* array to destroy.
- count – number of *N\_Vector* objects in vs array.

**Notes:**

This routine will internally call the *N\_Vector* implementation-specific *N\_VDestroy()* operation.

If vs was allocated using *N\_VCloneVectorArray()* then the data arrays for each *N\_Vector* object will be freed; if vs was allocated using *N\_VCloneVectorArrayEmpty()* then it is the user's responsibility to free the data for each *N\_Vector* object.

Finally, we note that users of the Fortran 2003 interface may be interested in the additional utility functions *N\_VNewVectorArray()*, *N\_VGetVecAtIndexVectorArray()*, and *N\_VSetVecAtIndexVectorArray()*, that are wrapped as *FN\_NewVectorArray*, *FN\_VGetVecAtIndexVectorArray*, and *FN\_VSetVecAtIndexVectorArray*, respectively. These functions allow a Fortran 2003 user to create an empty vector array, access a vector from this array, and set a vector within this array:

*N\_Vector* \***N\_VNewVectorArray**(int count, *SUNContext* sunctx)

Creates an array of count *N\_Vector* objects, the pointers to each are initialized as NULL.

**Arguments:**

- count – length of desired *N\_Vector* array.
- sunctx – a *SUNContext* object

**Return value:**

- pointer to a new *N\_Vector* array on success.
- NULL pointer on failure.

Changed in version 7.0.0: The function signature was updated to add the *SUNContext* argument.

*N\_Vector* \***N\_VGetVecAtIndexVectorArray**(*N\_Vector* \*vs, int index)

Accesses the *N\_Vector* at the location index within the *N\_Vector* array vs.

**Arguments:**

- vs – *N\_Vector* array.



- `index` – desired `N_Vector` to access from within `vs`.

**Return value:**

- pointer to the indexed `N_Vector` on success.
- NULL pointer on failure (`index < 0` or `vs == NULL`).

**Notes:**

This routine does not verify that `index` is within the extent of `vs`, since `vs` is a simple `N_Vector` array that does not internally store its allocated length.

void **N\_VSetVecAtIndexVectorArray**(*N\_Vector* \*vs, int index, *N\_Vector* w)

Sets a pointer to `w` at the location `index` within the vector array `vs`.

**Arguments:**

- `vs` – `N_Vector` array.
- `index` – desired location to place the pointer to `w` within `vs`.
- `w` – `N_Vector` to set within `vs`.

**Notes:**

This routine does not verify that `index` is within the extent of `vs`, since `vs` is a simple `N_Vector` array that does not internally store its allocated length.

## 8.1.2 Implementing a custom NVECTOR

A particular implementation of the NVECTOR module must:

- Specify the *content* field of the `N_Vector` structure.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly-defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly-defined `N_Vector`.

To aid in the creation of custom NVECTOR modules, the generic NVECTOR module provides two utility functions *N\_VNewEmpty()* and *N\_VCopyOps()*. When used in custom NVECTOR constructors and clone routines these functions will ease the introduction of any new optional vector operations to the NVECTOR API by ensuring that only required operations need to be set, and that all operations are copied when cloning a vector.

*N\_Vector* **N\_VNewEmpty**(*SUNContext* sunctx)

This allocates a new generic `N_Vector` object and initializes its content pointer and the function pointers in the operations structure to NULL.

**Return value:** If successful, this function returns an `N_Vector` object. If an error occurs when allocating the object, then this routine will return NULL.

void **N\_VFreeEmpty**(*N\_Vector* v)

This routine frees the generic `N_Vector` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the *ops* pointer is NULL, and, if it is not, it will free it as well.

**Arguments:**

- $v$  – an `N_Vector` object

*SUNErrCode* **N\_VCopyOps**(*N\_Vector*  $w$ , *N\_Vector*  $v$ )

This function copies the function pointers in the ops structure of  $w$  into the ops structure of  $v$ .

**Arguments:**

- $w$  – the vector to copy operations from
- $v$  – the vector to copy operations to

**Return value:** Returns a *SUNErrCode*.

enum **N\_Vector\_ID**

Each *N\_Vector* implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 8.1. It is recommended that a user supplied NVECTOR implementation use the SUNDIALS\_NVEC\_CUSTOM identifier.

Table 8.1: 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	<i>hypre</i> ParHyp parallel vector	4
SUNDIALS_NVEC_PETSC	PETSc parallel vector	5
SUNDIALS_NVEC_CUDA	CUDA vector	6
SUNDIALS_NVEC_HIP	HIP vector	7
SUNDIALS_NVEC_SYCL	SYCL vector	8
SUNDIALS_NVEC_RAJA	RAJA vector	9
SUNDIALS_NVEC_OPENMPDEV	OpenMP vector with device offloading	10
SUNDIALS_NVEC_TRILINOS	Trilinos Tpetra vector	11
SUNDIALS_NVEC_MANYVECTOR	“ManyVector” vector	12
SUNDIALS_NVEC_MPIMANYVECTOR	MPI-enabled “ManyVector” vector	13
SUNDIALS_NVEC_MPIPLUSX	MPI+X vector	14
SUNDIALS_NVEC_CUSTOM	User-provided custom vector	15

### 8.1.3 Support for complex-valued vectors

While SUNDIALS itself is written under an assumption of real-valued data, it does provide limited support for complex-valued problems. However, since none of the built-in NVECTOR modules supports complex-valued data, users must provide a custom NVECTOR implementation for this task. Many of the NVECTOR routines described in the subsection §8.2 naturally extend to complex-valued vectors; however, some do not. To this end, we provide the following guidance:

- *N\_VMin()* and *N\_VMinLocal()* should return the minimum of all *real* components of the vector, i.e.,  $m = \min_{0 \leq i < n} \text{real}(x_i)$ .
- *N\_VConst()* (and similarly *N\_VConstVectorArray()*) should set the real components of the vector to the input constant, and set all imaginary components to zero, i.e.,  $z_i = c + 0j$  for  $0 \leq i < n$ .
- *N\_VAddConst()* should only update the real components of the vector with the input constant, leaving all imaginary components unchanged.

- `N_VWrmsNorm()`, `N_VWrmsNormMask()`, `N_VWSqrSumLocal()` and `N_VWSqrSumMaskLocal()` should assume that all entries of the weight vector `w` and the mask vector `id` are real-valued.
- `N_VDotProd()` should mathematically return a complex number for complex-valued vectors; as this is not possible with SUNDIALS' current `sunrealtype`, this routine should be set to NULL in the custom NVECTOR implementation.
- `N_VCompare()`, `N_VConstrMask()`, `N_VMinQuotient()`, `N_VConstrMaskLocal()` and `N_VMinQuotientLocal()` are ill-defined due to the lack of a clear ordering in the complex plane. These routines should be set to NULL in the custom NVECTOR implementation.

While many SUNDIALS solver modules may be utilized on complex-valued data, others cannot. Specifically, although each package's linear solver interface (e.g., ARKLS or CVLS) may be used on complex-valued problems, none of the built-in SUNMatrix or SUNLinearSolver modules will work (all of the direct linear solvers must store complex-valued data, and all of the iterative linear solvers require `N_VDotProd()`). Hence a complex-valued user must provide custom linear solver modules for their problem. At a minimum this will consist of a custom SUNLinearSolver implementation (see §10.1.8), and optionally a custom SUNMatrix as well. The user should then attach these modules as normal to the package's linear solver interface.

Similarly, although both the `SUNNonlinearSolver_Newton` and `SUNNonlinearSolver_FixedPoint` modules may be used with any of the IVP solvers (CVODE(S), IDA(S) and ARKODE) for complex-valued problems, the Anderson-acceleration option with `SUNNonlinearSolver_FixedPoint` cannot be used due to its reliance on `N_VDotProd()`. By this same logic, the Anderson acceleration feature within KINSOL will also not work with complex-valued vectors.

Finally, constraint-handling features of each package cannot be used for complex-valued data, due to the issue of ordering in the complex plane discussed above with `N_VCompare()`, `N_VConstrMask()`, `N_VMinQuotient()`, `N_VConstrMaskLocal()` and `N_VMinQuotientLocal()`.

We provide a simple example of a complex-valued example problem, including a custom complex-valued Fortran 2003 NVECTOR module, in the files `examples/arkode/F2003_custom/ark_analytic_complex_f2003.f90`, `examples/arkode/F2003_custom/fnvector_complex_mod.f90`, and `examples/arkode/F2003_custom/test_fnvector_complex_mod.f90`.

## 8.2 Description of the NVECTOR operations

### 8.2.1 Standard vector operations

The standard vector operations defined by the generic `N_Vector` module are defined as follows. For each of these operations, we give the name, usage of the function, and a description of its mathematical operations below.

`N_Vector` **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 Table 8.1.

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 *sunrealtype* words
- *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);
```

Deprecated since version 7.3.0: Work space functions will be removed in version 8.0.0.

***sunrealtype* \*N\_VGetArrayPointer(*N\_Vector* v)**

Returns a pointer to a *sunrealtype* array from the *N\_Vector* *v*. Note that this assumes that the internal data in the *N\_Vector* is a contiguous array of *sunrealtype* and is accessible from the CPU.

This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.

Usage:

```
vdata = N_VGetArrayPointer(v);
```

***sunrealtype* \*N\_VGetDeviceArrayPointer(*N\_Vector* v)**

Returns a device pointer to a *sunrealtype* array from the *N\_Vector* *v*. Note that this assumes that the internal data in *N\_Vector* is a contiguous array of *sunrealtype* and is accessible from the device (e.g., GPU).

This operation is *optional* except when using the GPU-enabled direct linear solvers.

Usage:

```
vdata = N_VGetArrayPointer(v);
```

**void N\_VSetArrayPointer(*sunrealtype* \*vdata, *N\_Vector* v)**

Replaces the data array pointer in an *N\_Vector* with a given array of *sunrealtype*. Note that this assumes that the internal data in the *N\_Vector* is a contiguous array of *sunrealtype*. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module.

Usage:

```
N_VSetArrayPointer(vdata,v);
```

*SUNComm* **N\_VGetCommunicator**(*N\_Vector* v)

Returns the *SUNComm* (which is just an *MPI\_Comm* when SUNDIALS is built with MPI, otherwise it is an *int*) associated with the vector (if applicable). For MPI-unaware vector implementations, this should return *SUN\_COMM\_NULL*.

Usage:

```
MPI_Comm comm = N_VGetCommunicator(v); // Works if MPI is enabled
int comm = N_VGetCommunicator(v);      // Works if MPI is disabled
SUNComm comm = N_VGetCommunicator(v);  // Works with or without MPI
```

*sunindextype* **N\_VGetLength**(*N\_Vector* v)

Returns the global length (number of “active” entries) in the *NVECTOR* *v*. This value should be cumulative across all processes if the vector is used in a parallel environment. If *v* contains additional storage, e.g., for parallel communication, those entries should not be included.

Usage:

```
global_length = N_VGetLength(v);
```

*sunindextype* **N\_VGetLocalLength**(*N\_Vector* v)

Returns the local length (number of “active” entries) in the *NVECTOR* *v*. This value should be the length of the array returned by *N\_VGetArrayPointer()* or *N\_VGetDeviceArrayPointer()*.

Usage:

```
local_length = N_VGetLocalLength(v);
```

void **N\_VLinearSum**(*sunrealtype* a, *N\_Vector* x, *sunrealtype* b, *N\_Vector* y, *N\_Vector* z)

Performs the operation  $z = ax + by$ , where *a* and *b* are *sunrealtype* scalars and *x* and *y* are of type *N\_Vector*:

$$z_i = ax_i + by_i, \quad i = 0, \dots, n-1.$$

The output vector *z* can be the same as either of the input vectors (*x* or *y*).

Usage:

```
N_VLinearSum(a, x, b, y, z);
```

void **N\_VConst**(*sunrealtype* c, *N\_Vector* z)

Sets all components of the *N\_Vector* *z* to *sunrealtype* *c*:

$$z_i = c, \quad i = 0, \dots, n-1.$$

Usage:

```
N_VConst(c, z);
```

void **N\_VProd**(*N\_Vector* x, *N\_Vector* y, *N\_Vector* z)

Sets the *N\_Vector* *z* to be the component-wise product of the *N\_Vector* inputs *x* and *y*:

$$z_i = x_i y_i, \quad i = 0, \dots, n-1.$$

Usage:

```
N_VProd(x, y, z);
```

void **N\_VDiv**(*N\_Vector* x, *N\_Vector* y, *N\_Vector* z)

Sets the *N\_Vector*  $z$  to be the component-wise ratio of the *N\_Vector* inputs  $x$  and  $y$ :

$$z_i = \frac{x_i}{y_i}, \quad i = 0, \dots, n-1.$$

The  $y_i$  may not be tested for 0 values. It should only be called with a  $y$  that is guaranteed to have all nonzero components.

Usage:

```
N_VDiv(x, y, z);
```

void **N\_VScale**(*sunrealtype* c, *N\_Vector* x, *N\_Vector* z)

Scales the *N\_Vector*  $x$  by the *sunrealtype* scalar  $c$  and returns the result in  $z$ :

$$z_i = cx_i, \quad i = 0, \dots, n-1.$$

Usage:

```
N_VScale(c, x, z);
```

void **N\_VAbs**(*N\_Vector* x, *N\_Vector* z)

Sets the components of the *N\_Vector*  $z$  to be the absolute values of the components of the *N\_Vector*  $x$ :

$$z_i = |x_i|, \quad i = 0, \dots, n-1.$$

Usage:

```
N_VAbs(x, z);
```

void **N\_VInv**(*N\_Vector* x, *N\_Vector* z)

Sets the components of the *N\_Vector*  $z$  to be the inverses of the components of the *N\_Vector*  $x$ :

$$z_i = \frac{1}{x_i}, \quad i = 0, \dots, n-1.$$

This routine may not check for division by 0. It should be called only with an  $x$  which is guaranteed to have all nonzero components.

Usage:

```
N_VInv(x, z);
```

void **N\_VAddConst**(*N\_Vector* x, *sunrealtype* b, *N\_Vector* z)

Adds the *sunrealtype* scalar  $b$  to all components of  $x$  and returns the result in the *N\_Vector*  $z$ :

$$z_i = x_i + b, \quad i = 0, \dots, n-1.$$

Usage:

```
N_VAddConst(x, b, z);
```

*sunrealtype* **N\_VDotProd**(*N\_Vector* x, *N\_Vector* z)

Returns the value of the dot-product of the vectors  $x$  and  $y$ :

$$d = \sum_{i=0}^{n-1} x_i y_i.$$

Usage:

```
d = N_VDotProd(x, y);
```

*sunrealtype* **N\_VMaxNorm**(*N\_Vector* x)

Returns the value of the  $l_\infty$  norm of the *N\_Vector*  $x$ :

$$m = \max_{0 \leq i < n} |x_i|.$$

Usage:

```
m = N_VMaxNorm(x);
```

*sunrealtype* **N\_VWrmsNorm**(*N\_Vector* x, *N\_Vector* w)

Returns the weighted root-mean-square norm of the *N\_Vector*  $x$  with (positive) *sunrealtype* weight vector  $w$ :

$$m = \sqrt{\left( \sum_{i=0}^{n-1} (x_i w_i)^2 \right) / n}$$

Usage:

```
m = N_VWrmsNorm(x, w);
```

*sunrealtype* **N\_VWrmsNormMask**(*N\_Vector* x, *N\_Vector* w, *N\_Vector* id)

Returns the weighted root mean square norm of the *N\_Vector*  $x$  with *sunrealtype* weight vector  $w$  built using only the elements of  $x$  corresponding to positive elements of the *N\_Vector*  $id$ :

$$m = \sqrt{\left( \sum_{i=0}^{n-1} (x_i w_i H(id_i))^2 \right) / n},$$

$$\text{where } H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}.$$

Usage:

```
m = N_VWrmsNormMask(x, w, id);
```

*sunrealtype* **N\_VMin**(*N\_Vector* x)

Returns the smallest element of the *N\_Vector*  $x$ :

$$m = \min_{0 \leq i < n} x_i.$$

Usage:

```
m = N_VMin(x);
```

*sunrealtype* **N\_VWL2Norm**(*N\_Vector* x, *N\_Vector* w)Returns the weighted Euclidean  $l_2$  norm of the *N\_Vector*  $x$  with *sunrealtype* weight vector  $w$ :

$$m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}.$$

Usage:

```
m = N_VWL2Norm(x, w);
```

*sunrealtype* **N\_VL1Norm**(*N\_Vector* x)Returns the  $l_1$  norm of the *N\_Vector*  $x$ :

$$m = \sum_{i=0}^{n-1} |x_i|.$$

Usage:

```
m = N_VL1Norm(x);
```

void **N\_VCompare**(*sunrealtype* c, *N\_Vector* x, *N\_Vector* z)Compares the components of the *N\_Vector*  $x$  to the *sunrealtype* scalar  $c$  and returns an *N\_Vector*  $z$  such that for all  $0 \leq i < n$ ,

$$z_i = \begin{cases} 1.0 & \text{if } |x_i| \geq c, \\ 0.0 & \text{otherwise} \end{cases}.$$

Usage:

```
N_VCompare(c, x, z);
```

*sunboolean* **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:

$$z_i = \frac{1}{x_i}, \quad i = 0, \dots, n-1.$$

This routine returns a boolean assigned to `SUNTRUE` if all components of  $x$  are nonzero (successful inversion) and returns `SUNFALSE` otherwise.

Usage:

```
t = N_VInvTest(x, z);
```

*sunboolean* **N\_VConstrMask**(*N\_Vector* c, *N\_Vector* x, *N\_Vector* m)Performs the following constraint tests based on the values in  $c_i$ :

$$\begin{aligned} x_i &> 0 & \text{if } c_i = 2, \\ x_i &\geq 0 & \text{if } c_i = 1, \\ x_i &< 0 & \text{if } c_i = -2, \\ x_i &\leq 0 & \text{if } c_i = -1. \end{aligned}$$

There is no constraint on  $x_i$  if  $c_i = 0$ . This routine returns a boolean assigned to `SUNFALSE` if any element failed the constraint test and assigned to `SUNTRUE` if all passed. It also sets a mask vector  $m$ , with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Usage:



```
t = N_VConstrMask(c, x, m);
```

*sunrealtype* **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  $d$ :

$$\min_{0 \leq i < n} \frac{\text{num}_i}{\text{denom}_i}.$$

A zero element in *denom* will be skipped. If no such quotients are found, then the large value `SUN_BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

Usage:

```
minq = N_VMinQuotient(num, denom);
```

## 8.2.2 Fused operations

The following fused vector operations are *optional*. These operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused vector operations as `NULL`, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

*SUNErrCode* **N\_VLinearCombination**(int nv, *sunrealtype* \*c, *N\_Vector* \*X, *N\_Vector* z)

This routine computes the linear combination of  $nv$  vectors with  $n$  elements:

$$z_i = \sum_{j=0}^{nv-1} c_j x_{j,i}, \quad i = 0, \dots, n-1,$$

where  $c$  is an array of  $nv$  scalars,  $x_j$  is a vector in the vector array  $X$ , and  $z$  is the output vector. If the output vector  $z$  is one of the vectors in  $X$ , then it *must* be the first vector in the vector array. The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VLinearCombination(nv, c, X, z);
```

*SUNErrCode* **N\_VScaleAddMulti**(int nv, *sunrealtype* \*c, *N\_Vector* x, *N\_Vector* \*Y, *N\_Vector* \*Z)

This routine scales and adds one vector to  $nv$  vectors with  $n$  elements:

$$z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \dots, nv-1 \quad i = 0, \dots, n-1,$$

where  $c$  is an array of scalars,  $x$  is a vector,  $y_j$  is a vector in the vector array  $Y$ , and  $z_j$  is an output vector in the vector array  $Z$ . The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VScaleAddMulti(nv, c, x, Y, Z);
```

*SUNErrCode* **N\_VDotProdMulti**(int nv, *N\_Vector* x, *N\_Vector* \*Y, *sunrealtype* \*d)

This routine computes the dot product of a vector with  $nv$  vectors having  $n$  elements:

$$d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \dots, nv-1,$$

where  $d$  is an array of scalars containing the computed dot products,  $x$  is a vector, and  $y_j$  is a vector the vector array  $Y$ . The operation returns a [SUNErrCode](#).

Usage:

```
retval = N_VDotProdMulti(nv, x, Y, d);
```

### 8.2.3 Vector array operations

The following vector array operations are also *optional*. As with the fused vector operations, these are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused or vector array operations as NULL, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

[SUNErrCode](#) **N\_VLinearSumVectorArray**(int nv, *sunrealtype* a, *N\_Vector* \*X, *sunrealtype* b, *N\_Vector* \*Y, *N\_Vector* \*Z)

This routine computes the linear sum of two vector arrays of  $nv$  vectors with  $n$  elements:

$$z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, nv-1,$$

where  $a$  and  $b$  are scalars,  $x_j$  and  $y_j$  are vectors in the vector arrays  $X$  and  $Y$  respectively, and  $z_j$  is a vector in the output vector array  $Z$ . The operation returns a [SUNErrCode](#).

Usage:

```
retval = N_VLinearSumVectorArray(nv, a, X, b, Y, Z);
```

[SUNErrCode](#) **N\_VScaleVectorArray**(int nv, *sunrealtype* \*c, *N\_Vector* \*X, *N\_Vector* \*Z)

This routine scales each element in a vector of  $n$  elements in a vector array of  $nv$  vectors by a potentially different constant:

$$z_{j,i} = c_j x_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, nv-1,$$

where  $c$  is an array of scalars,  $x_j$  is a vector in the vector array  $X$ , and  $z_j$  is a vector in the output vector array  $Z$ . The operation returns a [SUNErrCode](#).

Usage:

```
retval = N_VScaleVectorArray(nv, c, X, Z);
```

[SUNErrCode](#) **N\_VConstVectorArray**(int nv, *sunrealtype* c, *N\_Vector* \*Z)

This routine sets each element in a vector of  $n$  elements in a vector array of  $nv$  vectors to the same value:

$$z_{j,i} = c, \quad i = 0, \dots, n-1 \quad j = 0, \dots, nv-1,$$

where  $c$  is a scalar and  $z_j$  is a vector in the vector array  $Z$ . The operation returns a [SUNErrCode](#).

Usage:

```
retval = N_VConstVectorArray(nv, c, Z);
```

**SUNErrCode N\_VWrmsNormVectorArray**(int nv, *N\_Vector* \*X, *N\_Vector* \*W, *sunrealtype* \*m)

This routine computes the weighted root mean square norm of each vector in a vector array:

$$m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2 \right)^{1/2}, \quad j = 0, \dots, nv - 1,$$

where  $x_j$  is a vector in the vector array  $X$ ,  $w_j$  is a weight vector in the vector array  $W$ , and  $m$  is the output array of scalars containing the computed norms. The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VWrmsNormVectorArray(nv, X, W, m);
```

**SUNErrCode N\_VWrmsNormMaskVectorArray**(int nv, *N\_Vector* \*X, *N\_Vector* \*W, *N\_Vector* id, *sunrealtype* \*m)

This routine computes the masked weighted root mean square norm of each vector in a vector array:

$$m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i} H(id_i))^2 \right)^{1/2}, \quad j = 0, \dots, nv - 1,$$

where  $H(id_i) = 1$  if  $id_i > 0$  and is zero otherwise,  $x_j$  is a vector in the vector array  $X$ ,  $w_j$  is a weight vector in the vector array  $W$ ,  $id$  is the mask vector, and  $m$  is the output array of scalars containing the computed norms. The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);
```

**SUNErrCode N\_VScaleAddMultiVectorArray**(int nv, int nsum, *sunrealtype* \*c, *N\_Vector* \*X, *N\_Vector* \*\*YY, *N\_Vector* \*\*ZZ)

This routine scales and adds a vector array of  $nv$  vectors to  $nsum$  other vector arrays:

$$z_{k,j,i} = c_k x_{j,i} + y_{k,j,i}, \quad i = 0, \dots, n - 1 \quad j = 0, \dots, nv - 1, \quad k = 0, \dots, nsum - 1$$

where  $c$  is an array of scalars,  $x_j$  is a vector in the vector array  $X$ ,  $y_{k,j}$  is a vector in the array of vector arrays  $YY$ , and  $z_{k,j}$  is an output vector in the array of vector arrays  $ZZ$ . The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VScaleAddMultiVectorArray(nv, nsum, c, x, YY, ZZ);
```

**SUNErrCode N\_VLinearCombinationVectorArray**(int nv, int nsum, *sunrealtype* \*c, *N\_Vector* \*\*XX, *N\_Vector* \*Z)

This routine computes the linear combination of  $nsum$  vector arrays containing  $nv$  vectors:

$$z_{j,i} = \sum_{k=0}^{nsum-1} c_k x_{k,j,i}, \quad i = 0, \dots, n - 1 \quad j = 0, \dots, nv - 1,$$

where  $c$  is an array of scalars,  $x_{k,j}$  is a vector in array of vector arrays  $XX$ , and  $z_{j,i}$  is an output vector in the vector array  $Z$ . If the output vector array is one of the vector arrays in  $XX$ , it *must* be the first vector array in  $XX$ . The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VLinearCombinationVectorArray(nv, nsum, c, XX, Z);
```

## 8.2.4 Local reduction operations

The following local reduction operations are also *optional*. As with the fused and vector array operations, these are intended to reduce parallel communication on distributed memory systems. If a particular NVECTOR implementation defines one of the local reduction operations as NULL, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

*sunrealtype* **N\_VDotProdLocal**(*N\_Vector* x, *N\_Vector* y)

This routine computes the MPI task-local portion of the ordinary dot product of  $x$  and  $y$ :

$$d = \sum_{i=0}^{n_{local}-1} x_i y_i,$$

where  $n_{local}$  corresponds to the number of components in the vector on this MPI task (or  $n_{local} = n$  for MPI-unaware applications).

Usage:

```
d = N_VDotProdLocal(x, y);
```

*sunrealtype* **N\_VMaxNormLocal**(*N\_Vector* x)

This routine computes the MPI task-local portion of the maximum norm of the NVECTOR  $x$ :

$$m = \max_{0 \leq i < n_{local}} |x_i|,$$

where  $n_{local}$  corresponds to the number of components in the vector on this MPI task (or  $n_{local} = n$  for MPI-unaware applications).

Usage:

```
m = N_VMaxNormLocal(x);
```

*sunrealtype* **N\_VMinLocal**(*N\_Vector* x)

This routine computes the smallest element of the MPI task-local portion of the NVECTOR  $x$ :

$$m = \min_{0 \leq i < n_{local}} x_i,$$

where  $n_{local}$  corresponds to the number of components in the vector on this MPI task (or  $n_{local} = n$  for MPI-unaware applications).

Usage:

```
m = N_VMinLocal(x);
```

*sunrealtype* **N\_VL1NormLocal**(*N\_Vector* x)

This routine computes the MPI task-local portion of the  $l_1$  norm of the N\_Vector  $x$ :

$$n = \sum_{i=0}^{n_{local}-1} |x_i|,$$

where  $n_{local}$  corresponds to the number of components in the vector on this MPI task (or  $n_{local} = n$  for MPI-unaware applications).

Usage:

```
n = N_VL1NormLocal(x);
```

*sunrealtype* **N\_VWSqrSumLocal**(*N\_Vector* x, *N\_Vector* w)

This routine computes the MPI task-local portion of the weighted squared sum of the NVECTOR *x* with weight vector *w*:

$$s = \sum_{i=0}^{n_{local}-1} (x_i w_i)^2,$$

where  $n_{local}$  corresponds to the number of components in the vector on this MPI task (or  $n_{local} = n$  for MPI-unaware applications).

Usage:

```
s = N_VWSqrSumLocal(x, w);
```

*sunrealtype* **N\_VWSqrSumMaskLocal**(*N\_Vector* x, *N\_Vector* w, *N\_Vector* id)

This routine computes the MPI task-local portion of the weighted squared sum of the NVECTOR *x* with weight vector *w* built using only the elements of *x* corresponding to positive elements of the NVECTOR *id*:

$$m = \sum_{i=0}^{n_{local}-1} (x_i w_i H(id_i))^2,$$

where

$$H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}$$

and  $n_{local}$  corresponds to the number of components in the vector on this MPI task (or  $n_{local} = n$  for MPI-unaware applications).

Usage:

```
s = N_VWSqrSumMaskLocal(x, w, id);
```

*sunbooleantype* **N\_VInvTestLocal**(*N\_Vector* x)

This routine sets the MPI task-local components of the NVECTOR *z* to be the inverses of the components of the NVECTOR *x*, with prior testing for zero values:

$$z_i = \frac{1}{x_i}, \quad i = 0, \dots, n_{local} - 1$$

where  $n_{local}$  corresponds to the number of components in the vector on this MPI task (or  $n_{local} = n$  for MPI-unaware applications). This routine returns a boolean assigned to SUNTRUE if all task-local components of *x* are nonzero (successful inversion) and returns SUNFALSE otherwise.

Usage:

```
t = N_VInvTestLocal(x);
```

*sunbooleantype* **N\_VConstrMaskLocal**(*N\_Vector* c, *N\_Vector* x, *N\_Vector* m)

Performs the following constraint tests based on the values in  $c_i$ :

$$\begin{array}{lll} x_i & > & 0 \quad \text{if } c_i = 2, \\ x_i & \geq & 0 \quad \text{if } c_i = 1, \\ x_i & < & 0 \quad \text{if } c_i = -2, \\ x_i & \leq & 0 \quad \text{if } c_i = -1. \end{array}$$

for all MPI task-local components of the vectors. This routine returns a boolean assigned to `SUNFALSE` if any task-local element failed the constraint test and assigned to `SUNTRUE` if all passed. It also sets a mask vector  $m$ , with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Usage:

```
t = N_VConstrMaskLocal(c, x, m);
```

*sunrealtype* **N\_VMinQuotientLocal**(*N\_Vector* num, *N\_Vector* denom)

This routine returns the minimum of the quotients obtained by term-wise dividing  $num_i$  by  $denom_i$ , for all MPI task-local components of the vectors. A zero element in  $denom$  will be skipped. If no such quotients are found, then the large value `SUN_BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

Usage:

```
minq = N_VMinQuotientLocal(num, denom);
```

## 8.2.5 Single Buffer Reduction Operations

The following *optional* operations are used to combine separate reductions into a single MPI call by splitting the local computation and communication into separate functions. These operations are used in low-synchronization orthogonalization methods to reduce the number of MPI `Allreduce` calls. If a particular `NVECTOR` implementation does not define these operations additional communication will be required.

*SUNErrCode* **N\_VDotProdMultiLocal**(int nv, *N\_Vector* x, *N\_Vector* \*Y, *sunrealtype* \*d)

This routine computes the MPI task-local portion of the dot product of a vector  $x$  with  $nv$  vectors  $y_j$ :

$$d_j = \sum_{i=0}^{n_{local}-1} x_i y_{j,i}, \quad j = 0, \dots, nv - 1,$$

where  $d$  is an array of scalars containing the computed dot products,  $x$  is a vector,  $y_j$  is a vector in the vector array  $Y$ , and  $n_{local}$  corresponds to the number of components in the vector on this MPI task. The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VDotProdMultiLocal(nv, x, Y, d);
```

*SUNErrCode* **N\_VDotProdMultiAllReduce**(int nv, *N\_Vector* x, *sunrealtype* \*d)

This routine combines the MPI task-local portions of the dot product of a vector  $x$  with  $nv$  vectors:

```
retval = MPI_Allreduce(MPI_IN_PLACE, d, nv, MPI_SUNREALTYPE, MPI_SUM, comm)
```

where  $d$  is an array of  $nv$  scalars containing the local contributions to the dot product and  $comm$  is the MPI communicator associated with the vector  $x$ . The operation returns a *SUNErrCode*.

Usage:

```
retval = N_VDotProdMultiAllReduce(nv, x, d);
```

## 8.2.6 Exchange operations

The following vector exchange operations are also *optional* and are intended only for use when interfacing with the XBraid library for parallel-in-time integration. In that setting these operations are required but are otherwise unused by SUNDIALS packages and may be set to NULL. For each operation, we give the function signature, a description of the expected behavior, and an example of the function usage.

*SUNErrCode* **N\_VBufSize**(*N\_Vector* x, *sunindextype* \*size)

This routine returns the buffer size need to exchange in the data in the vector *x* between computational nodes.

Usage:

```
flag = N_VBufSize(x, &buf_size)
```

*SUNErrCode* **N\_VBufPack**(*N\_Vector* x, void \*buf)

This routine fills the exchange buffer *buf* with the vector data in *x*.

Usage:

```
flag = N_VBufPack(x, &buf)
```

*SUNErrCode* **N\_VBufUnpack**(*N\_Vector* x, void \*buf)

This routine unpacks the data in the exchange buffer *buf* into the vector *x*.

Usage:

```
flag = N_VBufUnpack(x, buf)
```

## 8.2.7 Output operations

The following optional vector operations are for writing vector data either to stdout or to a given file.

void **N\_VPrint**(*N\_Vector* x)

This routine prints vector data to stdout

Usage:

```
N_VPrint(x);
```

void **N\_VPrintFile**(*N\_Vector* x, FILE \*file)

This routine writes vector data to the given file pointer.

Usage:

```
FILE* fp = fopen("vector_data.txt", "w");
N_VPrintFile(x, fp);
fclose(fp);
```

## 8.3 NVECTOR functions required by ARKODE

In Table 8.2 below, we list the vector functions in the *N\_Vector* module that are called within the ARKODE package. The table also shows, for each function, which ARKODE module uses the function. The ARKStep, ERKStep, MRISStep,

and SPRKStep columns show function usage within the main time-stepping modules and the shared ARKODE infrastructure, while the remaining columns show function usage within the ARKLS linear solver interface, within constraint-handling (i.e., when *ARKStepSetConstraints()* and *ERKStepSetConstraints()* are used), relaxation (i.e., when *ARKStepSetRelaxFn()*, *ERKStepSetRelaxFn()* and related are used), the ARKBANDPRE and ARKBBDPRE preconditioner modules.

Note that for ARKLS we only list the *N\_Vector* routines used directly by ARKLS, each *SUNLinearSolver* module may have additional requirements that are not listed here. In addition, specific *SUNNonlinearSolver* modules attached to ARKODE may have additional *N\_Vector* requirements. For additional requirements by specific *SUNLinearSolver* and *SUNNonlinearSolver* modules, please see the accompanying sections §10 and §11.

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 §8.1 are not listed in the above table. Therefore a user-supplied *N\_Vector* module for ARKODE could safely omit these functions from their implementation (although some may be needed by *SUNNonlinearSolver* or *SUNLinearSolver* modules).

Table 8.2: List of vector functions usage by ARKODE code modules

Routine	ARK- Step	ERK- Step	MRIS- tep	SPRK- Step	ARKLS Con- straints	Relax- ation	BBDPRE, BANDPRE
<i>N_VAbs()</i>	X	X	X	X			
<i>N_VAddConst()</i>	X	X	X	X			
<i>N_VClone()</i>	X	X	X	X	X		
<i>N_VCloneEmpty()</i>					1		
<i>N_VConst()</i>	X	X	X	X	X		
<i>N_VConstrMask()</i>						X	
<i>N_VDestroy()</i>	X	X	X	X	X		
<i>N_VDiv()</i>	X	X					
<i>N_VDotProd()</i>						X	
<i>N_VGetArray- Pointer()</i>					1		X
<i>N_VGetLength()</i>					4		
<i>N_VInv()</i>	X	X	X	X			
<i>N_VLinearCombi- nation()</i> <sup>3</sup>	X	X	X	X			
<i>N_VLinearSum()</i>	X	X	X	X	X	X	
<i>N_VMaxNorm()</i>	X	X				X	
<i>N_VMin()</i>	X	X	X	X			
<i>N_VMinQuotient()</i>						X	
<i>N_VProd()</i>						X	
<i>N_VScale()</i>	X	X	X	X	X	X	X
<i>N_VSetArray- Pointer()</i>					1		
<i>N_VSpace()</i> <sup>2</sup>	X	X	X	X	X		X
<i>N_VWrmsNorm()</i>	X	X	X	X	X		X

Special cases (numbers match markings in table):

1. This is only required with the *SUNMATRIX\_DENSE* or *SUNMATRIX\_BAND* modules, where the default difference-quotient Jacobian approximation is used.
2. The *N\_VSpace()* function is only informational, and will only be called if provided by the *N\_Vector* implementation.



3. The `N_VLinearCombination()` function is in fact optional; if it is not supplied then `N_VLinearSum()` will be used instead.
4. The `N_VGetLength()` function is only required when an iterative or matrix iterative SUNLinearSolver module is used.

## 8.4 The NVECTOR\_SERIAL Module

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR\_SERIAL, defines the *content* field of an `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;
    sunbooleantype own_data;
    sunrealtype *data;
};
```

The header file to be included when using this module is `nvector_serial.h`. The installed module library to link to is `libsundials_nvecserial.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

### 8.4.1 NVECTOR\_SERIAL accessor macros

The following five macros are provided to access the content of an NVECTOR\_SERIAL vector. The suffix `_S` in the names denotes the serial version.

#### NV\_CONTENT\_S(v)

This macro gives access to the contents of the serial vector `N_Vector v`.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector content` structure.

Implementation:

```
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

#### NV\_OWN\_DATA\_S(v)

Access the *own\_data* component of the serial `N_Vector v`.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
```

#### NV\_DATA\_S(v)

The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the *data* for the `N_Vector v`.

Similarly, the assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
```

**NV\_LENGTH\_S(v)**

Access the *length* component of the serial `N_Vector` `v`.

The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the *length* of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the *length* of `v` to be `len_v`.

Implementation:

```
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

**NV\_Ith\_S(v, i)**

This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

The assignment `r = NV_Ith_S(v, i)` sets `r` to be the value of the *i*-th component of `v`.

The assignment `NV_Ith_S(v, i) = r` sets the value of the *i*-th component of `v` to be `r`.

Here *i* ranges from 0 to  $n - 1$  for a vector of length *n*.

Implementation:

```
#define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
```

## 8.4.2 NVECTOR\_SERIAL functions

The `NVECTOR_SERIAL` module defines serial implementations of all vector operations listed in §8.2.1, §8.2.2, §8.2.3, and §8.2.4. Their names are obtained from those in those sections by appending the suffix `_Serial` (e.g. `N_VDestroy_Serial`). All the standard vector operations listed in §8.2.1 with the suffix `_Serial` appended are callable via the Fortran 2003 interface by prepending an `F` (e.g. `FN_VDestroy_Serial`).

The module `NVECTOR_SERIAL` provides the following additional user-callable routines:

`N_Vector N_VNew_Serial(sunindextype vec_length, SUNContext sunctx)`

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, SUNContext sunctx)`

This function creates a new serial `N_Vector` with an empty (NULL) data array.

`N_Vector N_VMake_Serial(sunindextype vec_length, sunrealtype *v_data, SUNContext sunctx)`

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.)

`void N_VPrint_Serial(N_Vector v)`

This function prints the content of a serial vector to `stdout`.

`void N_VPrintFile_Serial(N_Vector v, FILE *outfile)`

This function prints the content of a serial vector to `outfile`.

By default all fused and vector array operations are disabled in the `NVECTOR_SERIAL` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Serial()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees that the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned, from while vectors created with `N_VNew_Serial()` will have the default settings for the `NVECTOR_SERIAL` module.

**SUNErrCode N\_VEnableFusedOps\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableLinearCombination\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableScaleAddMulti\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableDotProdMulti\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableLinearSumVectorArray\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableScaleVectorArray\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableConstVectorArray\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableWrmsNormVectorArray\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableWrmsNormMaskVectorArray\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableScaleAddMultiVectorArray\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the serial vector. The return value is a *SUNErrCode*.

**SUNErrCode N\_VEnableLinearCombinationVectorArray\_Serial**(*N\_Vector* v, *sunboolean*type tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the serial vector. The return value is a *SUNErrCode*.

## 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)`, or equivalently `v_data = N_VGetArrayPointer(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()` and `N_VMake_Serial()` set the field `own_data` to `SUNFALSE`. The implementation of `N_VDestroy()` 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.

### 8.4.3 NVECTOR\_SERIAL Fortran Interface

The NVECTOR\_SERIAL module provides a Fortran 2003 module for use from Fortran applications.

The `fnvector_serial_mod` Fortran module defines interfaces to all NVECTOR\_SERIAL C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading F. For example, the function `N_VNew_Serial` is interfaced as `FN_VNew_Serial`.

The Fortran 2003 NVECTOR\_SERIAL interface module can be accessed with the `use` statement, i.e. `use fnvector_serial_mod`, and linking to the library `libsundials_fnvectorserial_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_serial_mod.mod` are installed see §17. We note that the module is accessible from the Fortran 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fnvectorserial_mod` library.

## 8.5 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 an `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;  
    sunboolean_t own_data;  
    sunrealtype *data;  
    MPI_Comm comm;  
};
```

The header file to be included when using this module is `nvector_parallel.h`. The installed module library to link to is `libsundials_nvecparallel.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

### 8.5.1 NVECTOR\_PARALLEL accessor macros

The following seven macros are provided to access the content of a NVECTOR\_PARALLEL vector. The suffix `_P` in the names denotes the distributed memory parallel version.

#### NV\_CONTENT\_P(v)

This macro gives access to the contents of the parallel `N_Vector` *v*.

The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` *content* structure of type `struct N_VectorContent_Parallel`.

Implementation:

```
#define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
```

#### NV\_OWN\_DATA\_P(v)

Access the *own\_data* component of the parallel `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_P(v) ( NV_CONTENT_P(v)->own_data )
```

**NV\_DATA\_P(v)**

The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the *local\_data* for the `N_Vector v`.

The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data` into *data*.

Implementation:

```
#define NV_DATA_P(v)      ( NV_CONTENT_P(v)->data )
```

**NV\_LOCLENGTH\_P(v)**

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`.

The call `NV_LOCLENGTH_P(v) = llen_v` sets the *local\_length* of `v` to be `llen_v`.

Implementation:

```
#define NV_LOCLENGTH_P(v) ( NV_CONTENT_P(v)->local_length )
```

**NV\_GLOBLENGTH\_P(v)**

The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the *global\_length* of the vector `v`.

The call `NV_GLOBLENGTH_P(v) = glen_v` sets the *global\_length* of `v` to be `glen_v`.

Implementation:

```
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

**NV\_COMM\_P(v)**

This macro provides access to the MPI communicator used by the parallel `N_Vector v`.

Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

**NV\_Ith\_P(v, i)**

This macro gives access to the individual components of the *local\_data* array of an `N_Vector`.

The assignment `r = NV_Ith_P(v, i)` sets `r` to be the value of the *i*-th component of the local part of `v`.

The assignment `NV_Ith_P(v, i) = r` sets the value of the *i*-th component of the local part of `v` to be `r`.

Here *i* ranges from 0 to  $n - 1$ , where *n* is the *local\_length*.

Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

**8.5.2 NVECTOR\_PARALLEL functions**

The `NVECTOR_PARALLEL` module defines parallel implementations of all vector operations listed in §8.2. Their names are obtained from the generic names 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, *SUNContext* sunctx)

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, *SUNContext* sunctx)

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, *sunrealtype* \*v\_data, *SUNContext* sunctx)

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.)

*sunindextype* **N\_VGetLocalLength\_Parallel**(*N\_Vector* v)

This function returns the local vector length.

void **N\_VPrint\_Parallel**(*N\_Vector* v)

This function prints the local content of a parallel vector to stdout.

void **N\_VPrintFile\_Parallel**(*N\_Vector* v, FILE \*outfile)

This function prints the local content of a parallel vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_PARALLEL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N\_VNew\_Parallel()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N\_VClone()*. This guarantees that the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from, while vectors created with *N\_VNew\_Parallel()* will have the default settings for the NVECTOR\_PARALLEL module.

*SUNErrCode* **N\_VEnableFusedOps\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_Parallel**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parallel vector. The return value is a *SUNErrCode*.



*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_Parallel**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_Parallel**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parallel vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_Parallel**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parallel vector. The return value is a *SUNErrCode*.

#### 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 = N_VGetArrayPointer(v)`, or equivalently `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()* and *N\_VMake\_Parallel()* set the field *own\_data* to SUNFALSE. The implementation of *N\_VDestroy()* 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.

### 8.5.3 NVECTOR\_PARALLEL Fortran Interface

The NVECTOR\_PARALLEL module provides a Fortran 2003 module for use from Fortran applications.

The `fnvector_parallel_mod` Fortran module defines interfaces to all NVECTOR\_PARALLEL C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading F. For example, the function `N_VNew_Parallel` is interfaced as `FN_VNew_Parallel`.

The Fortran 2003 NVECTOR\_PARALLEL interface module can be accessed with the `use` statement, i.e. `use fnvector_parallel_mod`, and linking to the library `libsundials_fnvectorparallel_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_parallel_mod.mod` are installed see §17. We note that the module is accessible from the Fortran 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fnvectorparallel_mod` library.

## 8.6 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;  
    sunbooleantype own_data;  
    sunrealtype *data;  
    int num_threads;  
};
```

The header file to be included when using this module is `nvector_openmp.h`. The installed module library to link to is `libsundials_nvecopenmp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries. The Fortran module file to use when using the Fortran 2003 interface to this module is `fnvector_openmp_mod.mod`.

### 8.6.1 NVECTOR\_OPENMP accessor macros

The following six macros are provided to access the content of an `NVECTOR_OPENMP` vector. The suffix `_OMP` in the names denotes the OpenMP version.

#### **NV\_CONTENT\_OMP(v)**

This macro gives access to the contents of the OpenMP vector `N_Vector v`.

The assignment `v_cont = NV_CONTENT_OMP(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
```

#### **NV\_OWN\_DATA\_OMP(v)**

Access the `own_data` component of the OpenMP `N_Vector v`.

Implementation:

```
#define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
```

#### **NV\_DATA\_OMP(v)**

The assignment `v_data = NV_DATA_OMP(v)` sets `v_data` to be a pointer to the first component of the `data` for the `N_Vector v`.

Similarly, the assignment `NV_DATA_OMP(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
```

#### **NV\_LENGTH\_OMP(v)**

Access the `length` component of the OpenMP `N_Vector v`.

The assignment `v_len = NV_LENGTH_OMP(v)` sets `v_len` to be the `length` of `v`. On the other hand, the call `NV_LENGTH_OMP(v) = len_v` sets the `length` of `v` to be `len_v`.

Implementation:

```
#define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
```



**NV\_NUM\_THREADS\_OMP(v)**

Access the *num\_threads* component of the OpenMP N\_Vector *v*.

The assignment `v_threads = NV_NUM_THREADS_OMP(v)` sets *v\_threads* to be the *num\_threads* of *v*. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the *num\_threads* of *v* to be *num\_threads\_v*.

Implementation:

```
#define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
```

**NV\_Ith\_OMP(v, i)**

This macro gives access to the individual components of the *data* array of an N\_Vector, using standard 0-based C indexing.

The assignment `r = NV_Ith_OMP(v, i)` sets *r* to be the value of the *i*-th component of *v*.

The assignment `NV_Ith_OMP(v, i) = r` sets the value of the *i*-th component of *v* to be *r*.

Here *i* ranges from 0 to  $n - 1$  for a vector of length *n*.

Implementation:

```
#define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
```

**8.6.2 NVECTOR\_OPENMP functions**

The NVECTOR\_OPENMP module defines OpenMP implementations of all vector operations listed in §8.2, §8.2.2, §8.2.3, and §8.2.4. Their names are obtained from those in those sections by appending the suffix `_OpenMP` (e.g. `N_VDestroy_OpenMP`). All the standard vector operations listed in §8.2 with the suffix `_OpenMP` appended are callable via the Fortran 2003 interface by prepending an *F*' (e.g. `FN_VDestroy_OpenMP`).

The module NVECTOR\_OPENMP provides the following additional user-callable routines:

*N\_Vector* **N\_VNew\_OpenMP**(*sunindextype* vec\_length, int num\_threads, *SUNContext* sunctx)

This function creates and allocates memory for an OpenMP N\_Vector. Arguments are the vector length and number of threads.

*N\_Vector* **N\_VNewEmpty\_OpenMP**(*sunindextype* vec\_length, int num\_threads, *SUNContext* sunctx)

This function creates a new OpenMP N\_Vector with an empty (NULL) data array.

*N\_Vector* **N\_VMake\_OpenMP**(*sunindextype* vec\_length, *sunrealtype* \*v\_data, int num\_threads, *SUNContext* sunctx)

This function creates and allocates memory for an OpenMP vector with user-provided data array, *v\_data*.

(This function does *not* allocate memory for *v\_data* itself.)

void **N\_VPrint\_OpenMP**(*N\_Vector* v)

This function prints the content of an OpenMP vector to stdout.

void **N\_VPrintFile\_OpenMP**(*N\_Vector* v, FILE \*outfile)

This function prints the content of an OpenMP vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_OPENMP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_OpenMP()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_OpenMP()` will have the default settings for the NVECTOR\_OPENMP module.

*SUNErrCode* **N\_VEnableFusedOps\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the OpenMP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_OpenMP**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the OpenMP vector. The return value is a *SUNErrCode*.

## Notes

- When looping over the components of an *N\_Vector* v, it is more efficient to first obtain the component array via `v_data = N_VGetArrayPointer(v)`, or equivalently `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()` and `N_VMake_OpenMP()` set the field `own_data` to `SUNFALSE`. The implementation of `N_VDestroy()` 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.

### 8.6.3 NVECTOR\_OPENMP Fortran Interface

The NVECTOR\_OPENMP module provides a Fortran 2003 module for use from Fortran applications.

The `fnvector_openmp_mod` Fortran module defines interfaces to all NVECTOR\_OPENMP C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading `F`. For example, the function `N_VNew_OpenMP` is interfaced as `FN_VNew_OpenMP`.

The Fortran 2003 NVECTOR\_OPENMP interface module can be accessed with the `use` statement, i.e. `use fnvector_openmp_mod`, and linking to the library `libsundials_fnvectoropenmp_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_openmp_mod.mod` are installed see §17.

## 8.7 The NVECTOR\_PTHREADS Module

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR\_OPENMP, and an implementation using Pthreads, called NVECTOR\_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, denoted NVECTOR\_PTHREADS, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own\_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
    sunindextype length;
    sunbooleantype own_data;
    sunrealtype *data;
    int num_threads;
};
```

The header file to be included when using this module is `nvector_pthreads.h`. The installed module library to link to is `libsundials_nvecpthreads.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

### 8.7.1 NVECTOR\_PTHREADS accessor macros

The following six macros are provided to access the content of an NVECTOR\_PTHREADS vector. The suffix `_PT` in the names denotes the Pthreads version.

#### NV\_CONTENT\_PT(v)

This macro gives access to the contents of the Pthreads vector `N_Vector v`.

The assignment `v_cont = NV_CONTENT_PT(v)` sets `v_cont` to be a pointer to the Pthreads `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
```

#### NV\_OWN\_DATA\_PT(v)

Access the *own\_data* component of the Pthreads `N_Vector v`.

Implementation:

```
#define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
```

#### NV\_DATA\_PT(v)

The assignment `v_data = NV_DATA_PT(v)` sets `v_data` to be a pointer to the first component of the *data* for the `N_Vector v`.

Similarly, the assignment `NV_DATA_PT(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
```

#### NV\_LENGTH\_PT(v)

Access the *length* component of the Pthreads `N_Vector v`.

The assignment `v_len = NV_LENGTH_PT(v)` sets `v_len` to be the *length* of `v`. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the *length* of `v` to be `len_v`.

Implementation:

```
#define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
```

#### NV\_NUM\_THREADS\_PT(v)

Access the *num\_threads* component of the Pthreads `N_Vector v`.

The assignment `v_threads = NV_NUM_THREADS_PT(v)` sets `v_threads` to be the *num\_threads* of `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the *num\_threads* of `v` to be `num_threads_v`.

Implementation:

```
#define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
```

#### NV\_Ith\_PT(v, i)

This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

The assignment `r = NV_Ith_PT(v, i)` sets `r` to be the value of the *i*-th component of `v`.

The assignment `NV_Ith_PT(v, i) = r` sets the value of the *i*-th component of `v` to be `r`.

Here *i* ranges from 0 to  $n - 1$  for a vector of length *n*.

Implementation:

```
#define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
```

## 8.7.2 NVECTOR\_PTHREADS functions

The `NVECTOR_PTHREADS` module defines Pthreads implementations of all vector operations listed in §8.2, §8.2.2, §8.2.3, and §8.2.4. Their names are obtained from those in those sections by appending the suffix `_Pthreads` (e.g. `N_VDestroy_Pthreads`). All the standard vector operations listed in §8.2 are callable via the Fortran 2003 interface by prepending an *F* (e.g. `FN_VDestroy_Pthreads`). The module `NVECTOR_PTHREADS` provides the following additional user-callable routines:

*N\_Vector* **N\_VNew\_Pthreads**(*sunindextype* vec\_length, int num\_threads, *SUNContext* sunctx)

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, *SUNContext* sunctx)

This function creates a new Pthreads *N\_Vector* with an empty (NULL) data array.

*N\_Vector* **N\_VMake\_Pthreads**(*sunindextype* vec\_length, *sunrealtype* \*v\_data, int num\_threads, *SUNContext* sunctx)

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.)

void **N\_VPrint\_Pthreads**(*N\_Vector* v)

This function prints the content of a Pthreads vector to stdout.

void **N\_VPrintFile\_Pthreads**(*N\_Vector* v, FILE \*outfile)

This function prints the content of a Pthreads vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_PTHREADS module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N\_VNew\_Pthreads()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N\_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N\_VNew\_Pthreads()* will have the default settings for the NVECTOR\_PTHREADS module.

*SUNErrCode* **N\_VEnableFusedOps\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_Pthreads**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_Pthreads**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_Pthreads**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the Pthreads vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_Pthreads**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the Pthreads vector. The return value is a *SUNErrCode*.

## Notes

- When looping over the components of an *N\_Vector* v, it is more efficient to first obtain the component array via `v_data = N_VGetArrayPointer(v)`, or equivalently `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()* and *N\_VMake\_Pthreads()* set the field *own\_data* to SUNFALSE. The implementation of *N\_VDestroy()* 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.

## 8.7.3 NVECTOR\_PTHREADS Fortran Interface

The NVECTOR\_PTHREADS module provides a Fortran 2003 module for use from Fortran applications.

The `fnvector_pthreads_mod` Fortran module defines interfaces to all NVECTOR\_PTHREADS C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading F. For example, the function `N_VNew_Pthreads` is interfaced as `FN_VNew_Pthreads`.

The Fortran 2003 NVECTOR\_PTHREADS interface module can be accessed with the `use` statement, i.e. `use fn-vector_pthreads_mod`, and linking to the library `libsundials_fnvectorpthreads_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_pthreads_mod.mod` are installed see §17.

## 8.8 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;
    sunboolean own_data;
    sunboolean own_parvector;
    sunrealtype *data;
```

(continues on next page)



(continued from previous page)

```

MPI_Comm comm;
hypre_ParVector *x;
};

```

The header file to be included when using this module is `nvector_parhyp.h`. The installed module library to link to is `libsundials_nvecparhyp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native SUNDIALS vector types, `NVECTOR_PARHYP` does not provide macros to access its member variables. Note that `NVECTOR_PARHYP` requires SUNDIALS to be built with MPI support.

### 8.8.1 NVECTOR\_PARHYP functions

The `NVECTOR_PARHYP` module defines implementations of all vector operations listed in §8.2 except for `N_VSetArrayPointer()` and `N_VGetArrayPointer()` because accessing raw vector data is handled by low-level HYPRE functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract the HYPRE vector first, and then use HYPRE methods to access the data. Usage examples of `NVECTOR_PARHYP` are provided in the `cvAdvDiff_non_ph.c` example programs for CVODE and the `ark_diurnal_kry_ph.c` example program for ARKODE.

The names of parhyp methods are obtained from those in §8.2, §8.2.2, §8.2.3, and §8.2.4 by appending the suffix `_ParHyp` (e.g. `N_VDestroy_ParHyp`). The module `NVECTOR_PARHYP` provides the following additional user-callable routines:

*N\_Vector* **N\_VNewEmpty\_ParHyp**(MPI\_Comm comm, *sunindextype* local\_length, *sunindextype* global\_length, *SUNContext* sunctx)

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, *SUNContext* sunctx)

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.

void **N\_VPrint\_ParHyp**(*N\_Vector* v)

This function prints the local content of a parhyp vector to `stdout`.

void **N\_VPrintFile\_ParHyp**(*N\_Vector* v, FILE \*outfile)

This function prints the local content of a parhyp vector to `outfile`.

By default all fused and vector array operations are disabled in the `NVECTOR_PARHYP` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VMake_ParHyp()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VMake_ParHyp()` will have the default settings for the `NVECTOR_PARHYP` module.

*SUNErrCode* **N\_VEnableFusedOps\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parhyp vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_ParHyp**(*N\_Vector* v, *sunboolean* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parhyp vector. The return value is a *SUNErrCode*.

## 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()***, and ***N\_VMake\_ParHyp()*** set the field *own\_parvector* to *SUNFALSE*. The implementation of ***N\_VDestroy()*** 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.

## 8.9 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;
    sunbooleantype own_data;
    Vec *pvec;
    MPI_Comm comm;
};

```

The header file to be included when using this module is `nvector_petsc.h`. The installed module library to link to is `libsundials_nvecpetsc.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native SUNDIALS vector types, `NVECTOR_PETSC` does not provide macros to access its member variables. Note that `NVECTOR_PETSC` requires SUNDIALS to be built with MPI support.

### 8.9.1 NVECTOR\_PETSC functions

The `NVECTOR_PETSC` module defines implementations of all vector operations listed in §8.2 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 §8.2, §8.2.2, §8.2.3, and §8.2.4 by appending the suffix `_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, *SUNContext* sunctx)

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, *SUNContext* sunctx)

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.

void **N\_VPrint\_Petsc**(*N\_Vector* v)

This function prints the global content of a wrapped PETSc vector to `stdout`.

void **N\_VPrintFile\_Petsc**(*N\_Vector* v, const char fname[])

This function prints the global content of a wrapped PETSc vector to `fname`.

By default all fused and vector array operations are disabled in the `NVECTOR_PETSC` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VMake_Petsc()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VMake_Petsc()` will have the default settings for the `NVECTOR_PETSC` module.

*SUNErrCode* **N\_VEnableFusedOps\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the PETSc vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_Petsc**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the PETSc vector. The return value is a *SUNErrCode*.

## Notes

- When there is a need to access components of an *N\_Vector\_Petsc* v, it is recommended 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()* and *N\_VMake\_Petsc()*, set the field *own\_data* to *SUNFALSE*. The implementation of *N\_VDestroy()* 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.

## 8.10 The NVECTOR\_CUDA Module

The NVECTOR\_CUDA module is an NVECTOR implementation in the CUDA language. The module allows for SUNDIALS vector kernels to run on NVIDIA GPU devices. It is intended for users who are already familiar with CUDA and GPU programming. Building this vector module requires a CUDA compiler and, by extension, a C++ compiler. The vector content layout is as follows:

```
struct _N_VectorContent_Cuda
{
    sunindextype      length;
    sunboolean_type   own_helper;
    SUNMemory         host_data;
    SUNMemory         device_data;
    SUNCudaExecPolicy* stream_exec_policy;
    SUNCudaExecPolicy* reduce_exec_policy;
    SUNMemoryHelper   mem_helper;
    void*             priv; /* 'private' data */
};

typedef struct _N_VectorContent_Cuda *N_VectorContent_Cuda;
```

The content members are the vector length (size), boolean flags that indicate if the vector owns the execution policies and memory helper objects (i.e., it is in charge of freeing the objects), *SUNMemory* objects for the vector data on the host and device, pointers to execution policies that control how streaming and reduction kernels are launched, a *SUNMemoryHelper* for performing memory operations, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with *N\_VNew\_Cuda()*, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the *N\_VMake\_Cuda()* constructor. To use CUDA managed memory, the constructors *N\_VNewManaged\_Cuda()* and *N\_VMakeManaged\_Cuda()* are provided. Additionally, a user-defined *SUNMemoryHelper* for allocating/freeing data can be provided with the constructor *N\_VNewWithMemHelp\_Cuda()*. Details on each of these constructors are provided below.

To use the NVECTOR\_CUDA module, include *nvector\_cuda.h* and link to the library *libsundials\_nveccuda.lib*. The extension, *.lib*, is typically *.so* for shared libraries and *.a* for static libraries.

### 8.10.1 NVECTOR\_CUDA functions

Unlike other native SUNDIALS vector types, the NVECTOR\_CUDA module does not provide macros to access its member variables. Instead, user should use the accessor functions:

*sunrealtype* \**N\_VGetHostArrayPointer\_Cuda*(*N\_Vector* v)

This function returns pointer to the vector data on the host.

*sunrealtype* \**N\_VGetDeviceArrayPointer\_Cuda*(*N\_Vector* v)

This function returns pointer to the vector data on the device.

*sunboolean\_type* *N\_VIsManagedMemory\_Cuda*(*N\_Vector* v)

This function returns a boolean flag indicating if the vector data array is in managed memory or not.

The NVECTOR\_CUDA module defines implementations of all standard vector operations defined in §8.2, §8.2.2, §8.2.3, and §8.2.4, except for *N\_VSetArrayPointer()*, and, if using unmanaged memory, *N\_VGetArrayPointer()*. As such, this vector can only be used with SUNDIALS direct solvers and preconditioners when using managed memory. The NVECTOR\_CUDA module provides separate functions to access data on the host and on the device for the

unmanaged memory use case. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR\_CUDA are provided in example programs for CVOICE [62].

The names of vector operations are obtained from those in §8.2, §8.2.2, §8.2.3, and §8.2.4 by appending the suffix `_Cuda` (e.g. `N_VDestroy_Cuda`). The module NVECTOR\_CUDA provides the following additional user-callable routines:

`N_Vector N_VNew_Cuda`(*sunindextype* length, *SUNContext* sunctx)

This function creates and allocates memory for a CUDA `N_Vector`. The vector data array is allocated on both the host and device.

`N_Vector N_VNewManaged_Cuda`(*sunindextype* vec\_length, *SUNContext* sunctx)

This function creates and allocates memory for a CUDA `N_Vector`. The vector data array is allocated in managed memory.

`N_Vector N_VNewWithMemHelp_Cuda`(*sunindextype* length, *sunbooleantype* use\_managed\_mem, *SUNMemoryHelper* helper, *SUNContext* sunctx)

This function creates a new CUDA `N_Vector` with a user-supplied `SUNMemoryHelper` for allocating/freeing memory.

`N_Vector N_VNewEmpty_Cuda`(*sunindextype* vec\_length, *SUNContext* sunctx)

This function creates a new CUDA `N_Vector` where the members of the content structure have not been allocated. This utility function is used by the other constructors to create a new vector.

`N_Vector N_VMake_Cuda`(*sunindextype* vec\_length, *sunrealtype* \*h\_vdata, *sunrealtype* \*d\_vdata, *SUNContext* sunctx)

This function creates a CUDA `N_Vector` with user-supplied vector data arrays for the host and the device.

`N_Vector N_VMakeManaged_Cuda`(*sunindextype* vec\_length, *sunrealtype* \*vdata, *SUNContext* sunctx)

This function creates a CUDA `N_Vector` with a user-supplied managed memory data array.

`N_Vector N_VMakeWithManagedAllocator_Cuda`(*sunindextype* length, void \*(\*allocfn)(size\_t size), void (\*freefn)(void \*ptr))

This function creates a CUDA `N_Vector` with a user-supplied memory allocator. It requires the user to provide a corresponding free function as well. The memory allocated by the allocator function must behave like CUDA managed memory.

The module NVECTOR\_CUDA also provides the following user-callable routines:

void `N_VSetKernelExecPolicy_Cuda`(*N\_Vector* v, *SUNCudaExecPolicy* \*stream\_exec\_policy, *SUNCudaExecPolicy* \*reduce\_exec\_policy)

This function sets the execution policies which control the kernel parameters utilized when launching the stream-ing and reduction CUDA kernels. By default the vector is setup to use the `SUNCudaThreadDirectExecPolicy` and `SUNCudaBlockReduceAtomicExecPolicy`. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the CUDA warp size (32). See §8.10.2 below for more information about the `SUNCudaExecPolicy` class. Providing NULL for an argument will result in the default policy being restored.

The input execution policies are cloned and, as such, may be freed after being attached to the desired vectors. A NULL input policy will reset the execution policy to the default setting.

#### Note

Note: All vectors used in a single instance of a SUNDIALS package must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors

*sunrealtype* **\*N\_VCopyToDevice\_Cuda**(*N\_Vector* v)

This function copies host vector data to the device.

*sunrealtype* **\*N\_VCopyFromDevice\_Cuda**(*N\_Vector* v)

This function copies vector data from the device to the host.

void **N\_VPrint\_Cuda**(*N\_Vector* v)

This function prints the content of a CUDA vector to stdout.

void **N\_VPrintFile\_Cuda**(*N\_Vector* v, FILE \*outfile)

This function prints the content of a CUDA vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_CUDA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N\_VNew\_Cuda()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N\_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N\_VNew\_Cuda()* will have the default settings for the NVECTOR\_CUDA module.

*SUNErrCode* **N\_VEnableFusedOps\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the CUDA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_Cuda**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the CUDA vector. The return value is a *SUNErrCode*.

#### 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()*. However, when using managed memory, the function *N\_VGetArrayPointer()* may also be used.
- To maximize efficiency, vector operations in the NVECTOR\_CUDA implementation that have more than one *N\_Vector* argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with *N\_Vector* arguments that were all created with the same internal representations.

### 8.10.2 The SUNCudaExecPolicy Class

In order to provide maximum flexibility to users, the CUDA kernel execution parameters used by kernels within SUNDIALS are defined by objects of the `sundials::cuda::ExecPolicy` abstract class type (this class can be accessed in the global namespace as `SUNCudaExecPolicy`). Thus, users may provide custom execution policies that fit the needs of their problem. The `SUNCudaExecPolicy` class is defined as

```
typedef sundials::cuda::ExecPolicy SUNCudaExecPolicy
```

where the `sundials::cuda::ExecPolicy` class is defined in the header file `sundials_cuda_policies.hpp`, as follows:

```
class sundials::cuda::ExecPolicy
{
    ExecPolicy(cudaStream_t stream = 0)

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0)

    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0)

    virtual const cudaStream_t *stream() const

    virtual ExecPolicy *clone() const

    ExecPolicy *clone_new_stream(cudaStream_t stream) const

    virtual bool atomic() const

    virtual ~ExecPolicy()
}
```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::cuda::ThreadDirectExecPolicy` (aka in the global namespace as `SUNCudaThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class ThreadDirectExecPolicy : public ExecPolicy
{
public:
    ThreadDirectExecPolicy(const size_t blockDim, cudaStream_t stream = 0)
```

(continues on next page)



(continued from previous page)

```

        : blockDim_(blockDim), ExecPolicy(stream)
    {}

    ThreadDirectExecPolicy(const ThreadDirectExecPolicy& ex)
        : blockDim_(ex.blockDim_), ExecPolicy(ex.stream_)
    {}

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t /*blockDim*/ = 0) const
    {
        /* ceil(n/m) = floor((n + m - 1) / m) */
        return (numWorkUnits + blockSize() - 1) / blockSize();
    }

    virtual size_t blockSize(size_t /*numWorkUnits*/ = 0, size_t /*gridDim*/ = 0) const
    {
        return blockDim_;
    }

    virtual ExecPolicy* clone() const
    {
        return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
    }

private:
    const size_t blockDim_;
};

```

In total, SUNDIALS provides 3 execution policies:

**SUNCudaThreadDirectExecPolicy**(const size\_t blockDim, const cudaStream\_t stream = 0)

Maps each CUDA thread to a work unit. The number of threads per block (blockDim) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a CUDA stream is provided, it will be used to execute the kernel.

**SUNCudaGridStrideExecPolicy**(const size\_t blockDim, const size\_t gridDim, const cudaStream\_t stream = 0)

Is for kernels that use grid stride loops. The number of threads per block (blockDim) can be set to anything. The number of blocks (gridDim) can be set to anything. If a CUDA stream is provided, it will be used to execute the kernel.

**SUNCudaBlockReduceExecPolicy**(const size\_t blockDim, const cudaStream\_t stream = 0)

Is for kernels performing a reduction across individual thread blocks. The number of threads per block (blockDim) can be set to any valid multiple of the CUDA warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a CUDA stream is provided, it will be used to execute the kernel.

**SUNCudaBlockReduceAtomicExecPolicy**(const size\_t blockDim, const cudaStream\_t stream = 0)

Is for kernels performing a reduction across individual thread blocks using atomic operations. The number of threads per block (blockDim) can be set to any valid multiple of the CUDA warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a CUDA stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
SUNCudaThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures (e.g. a *SUNMatrix* and an *N\_Vector*) since they do not hold any modifiable state information.

## 8.11 The NVECTOR\_HIP Module

The NVECTOR\_HIP module is an NVECTOR implementation using the AMD ROCm HIP library [2]. The module allows for SUNDIALS vector kernels to run on AMD or NVIDIA GPU devices. It is intended for users who are already familiar with HIP and GPU programming. Building this vector module requires the HIP-clang compiler. The vector content layout is as follows:

```
struct _N_VectorContent_Hip
{
    sunindextype      length;
    sunboolean_t      own_helper;
    SUNMemory         host_data;
    SUNMemory         device_data;
    SUNHipExecPolicy* stream_exec_policy;
    SUNHipExecPolicy* reduce_exec_policy;
    SUNMemoryHelper    mem_helper;
    void*             priv; /* 'private' data */
};

typedef struct _N_VectorContent_Hip *N_VectorContent_Hip;
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the data (i.e. it is in charge of freeing the data), pointers to vector data on the host and the device, pointers to *SUNHipExecPolicy* implementations that control how the HIP kernels are launched for streaming and reduction vector kernels, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with *N\_VNew\_Hip()*, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the *N\_VMake\_Hip()* constructor. To use managed memory, the constructors *N\_VNewManaged\_Hip()* and *N\_VMakeManaged\_Hip()* are provided. Additionally, a user-defined *SUNMemoryHelper* for allocating/freeing data can be provided with the constructor *N\_VNewWithMemHelp\_Hip()*. Details on each of these constructors are provided below.

To use the NVECTOR\_HIP module, include *nvector\_hip.h* and link to the library *libsundials\_nvechip.lib*. The extension, *.lib*, is typically *.so* for shared libraries and *.a* for static libraries.

### 8.11.1 NVECTOR\_HIP functions

Unlike other native SUNDIALS vector types, the NVECTOR\_HIP module does not provide macros to access its member variables. Instead, user should use the accessor functions:

*sunrealtype* \**N\_VGetHostArrayPointer\_Hip*(*N\_Vector* v)

This function returns pointer to the vector data on the host.



*sunrealtype* \***N\_VGetDeviceArrayPointer\_Hip**(*N\_Vector* v)

This function returns pointer to the vector data on the device.

*sunbooleantype* **N\_VIsManagedMemory\_Hip**(*N\_Vector* v)

This function returns a boolean flag indicating if the vector data array is in managed memory or not.

The NVECTOR\_HIP module defines implementations of all standard vector operations defined in §8.2, §8.2.2, §8.2.3, and §8.2.4, except for *N\_VSetArrayPointer()*. The names of vector operations are obtained from those in §8.2, §8.2.2, §8.2.3, and §8.2.4 by appending the suffix *\_Hip* (e.g. *N\_VDestroy\_Hip*). The module NVECTOR\_HIP provides the following additional user-callable routines:

*N\_Vector* **N\_VNew\_Hip**(*sunindextype* length, *SUNContext* sunctx)

This function creates and allocates memory for a HIP *N\_Vector*. The vector data array is allocated on both the host and device.

*N\_Vector* **N\_VNewManaged\_Hip**(*sunindextype* vec\_length, *SUNContext* sunctx)

This function creates and allocates memory for a HIP *N\_Vector*. The vector data array is allocated in managed memory.

*N\_Vector* **N\_VNewWithMemHelp\_Hip**(*sunindextype* length, *sunbooleantype* use\_managed\_mem, *SUNMemoryHelper* helper, *SUNContext* sunctx)

This function creates a new HIP *N\_Vector* with a user-supplied *SUNMemoryHelper* for allocating/freeing memory.

*N\_Vector* **N\_VNewEmpty\_Hip**(*sunindextype* vec\_length, *SUNContext* sunctx)

This function creates a new HIP *N\_Vector* where the members of the content structure have not been allocated. This utility function is used by the other constructors to create a new vector.

*N\_Vector* **N\_VMake\_Hip**(*sunindextype* vec\_length, *sunrealtype* \*h\_vdata, *sunrealtype* \*d\_vdata, *SUNContext* sunctx)

This function creates a HIP *N\_Vector* with user-supplied vector data arrays for the host and the device.

*N\_Vector* **N\_VMakeManaged\_Hip**(*sunindextype* vec\_length, *sunrealtype* \*vdata, *SUNContext* sunctx)

This function creates a HIP *N\_Vector* with a user-supplied managed memory data array.

The module NVECTOR\_HIP also provides the following user-callable routines:

void **N\_VSetKernelExecPolicy\_Hip**(*N\_Vector* v, *SUNHipExecPolicy* \*stream\_exec\_policy, *SUNHipExecPolicy* \*reduce\_exec\_policy)

This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction HIP kernels. By default the vector is setup to use the *SUNHipThreadDirectExecPolicy()* and *SUNHipBlockReduceExecPolicy()*. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the HIP warp size (32 for NVIDIA GPUs, 64 for AMD GPUs). See §8.11.2 below for more information about the *SUNHipExecPolicy* class. Providing NULL for an argument will result in the default policy being restored.

The input execution policies are cloned and, as such, may be freed after being attached to the desired vectors. A NULL input policy will reset the execution policy to the default setting.

#### Note

Note: All vectors used in a single instance of a SUNDIALS package must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors\*

*sunrealtype* \*N\_VCopyToDevice\_Hip(*N\_Vector* v)

This function copies host vector data to the device.

*sunrealtype* \*N\_VCopyFromDevice\_Hip(*N\_Vector* v)

This function copies vector data from the device to the host.

void N\_VPrint\_Hip(*N\_Vector* v)

This function prints the content of a HIP vector to stdout.

void N\_VPrintFile\_Hip(*N\_Vector* v, FILE \*outfile)

This function prints the content of a HIP vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_HIP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N\_VNew\_Hip()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N\_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N\_VNew\_Hip()* will have the default settings for the NVECTOR\_HIP module.

*SUNErrCode* N\_VEnableFusedOps\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableLinearCombination\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableScaleAddMulti\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableDotProdMulti\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableLinearSumVectorArray\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableScaleVectorArray\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableConstVectorArray\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableWrmsNormVectorArray\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* N\_VEnableWrmsNormMaskVectorArray\_Hip(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_Hip**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the HIP vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_Hip**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the HIP vector. The return value is a *SUNErrCode*.

#### Notes

- When there is a need to access components of an *N\_Vector\_Hip*, v, it is recommended to use functions *N\_VGetDeviceArrayPointer\_Hip()* or *N\_VGetHostArrayPointer\_Hip()*. However, when using managed memory, the function *N\_VGetArrayPointer()* may also be used.
- To maximize efficiency, vector operations in the NVECTOR\_HIP implementation that have more than one *N\_Vector* argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with *N\_Vector* arguments that were all created with the same internal representations.

### 8.11.2 The SUNHipExecPolicy Class

In order to provide maximum flexibility to users, the HIP kernel execution parameters used by kernels within SUNDIALS are defined by objects of the `sundials::hip::ExecPolicy` abstract class type (this class can be accessed in the global namespace as `SUNHipExecPolicy`). Thus, users may provide custom execution policies that fit the needs of their problem. The `SUNHipExecPolicy` class is defined as

```
typedef sundials::hip::ExecPolicy SUNHipExecPolicy
```

where the `sundials::hip::ExecPolicy` class is defined in the header file `sundials_hip_policies.hpp`, as follows:

```
class sundials::hip::ExecPolicy
{
    ExecPolicy(hipStream_t stream = 0)

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0)

    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0)

    virtual const hipStream_t *stream() const

    virtual ExecPolicy *clone() const

    ExecPolicy *clone_new_stream(hipStream_t stream) const

    virtual bool atomic() const

    virtual ~ExecPolicy()
```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::hip::ThreadDirectExecPolicy` (aka in the global namespace as `SUNHipThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class ThreadDirectExecPolicy : public ExecPolicy
{
public:
    ThreadDirectExecPolicy(const size_t blockDim, hipStream_t stream = 0)
```

(continues on next page)

(continued from previous page)

```

        : blockDim_(blockDim), ExecPolicy(stream)
    {}

    ThreadDirectExecPolicy(const ThreadDirectExecPolicy& ex)
        : blockDim_(ex.blockDim_), ExecPolicy(ex.stream_)
    {}

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t /*blockDim*/ = 0) const
    {
        /* ceil(n/m) = floor((n + m - 1) / m) */
        return (numWorkUnits + blockSize() - 1) / blockSize();
    }

    virtual size_t blockSize(size_t /*numWorkUnits*/ = 0, size_t /*gridDim*/ = 0) const
    {
        return blockDim_;
    }

    virtual ExecPolicy* clone() const
    {
        return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
    }

private:
    const size_t blockDim_;
};

```

In total, SUNDIALS provides 4 execution policies:

**SUNHipThreadDirectExecPolicy**(const size\_t blockDim, const hipStream\_t stream = 0)

Maps each HIP thread to a work unit. The number of threads per block (blockDim) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a HIP stream is provided, it will be used to execute the kernel.

**SUNHipGridStrideExecPolicy**(const size\_t blockDim, const size\_t gridDim, const hipStream\_t stream = 0)

Is for kernels that use grid stride loops. The number of threads per block (blockDim) can be set to anything. The number of blocks (gridDim) can be set to anything. If a HIP stream is provided, it will be used to execute the kernel.

**SUNHipBlockReduceExecPolicy**(const size\_t blockDim, const hipStream\_t stream = 0)

Is for kernels performing a reduction across individual thread blocks. The number of threads per block (blockDim) can be set to any valid multiple of the HIP warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a HIP stream is provided, it will be used to execute the kernel.

**SUNHipBlockReduceAtomicExecPolicy**(const size\_t blockDim, const hipStream\_t stream = 0)

Is for kernels performing a reduction across individual thread blocks using atomic operations. The number of threads per block (blockDim) can be set to any valid multiple of the HIP warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a HIP stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
hipStream_t stream;
hipStreamCreate(&stream);
SUNHipThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures (e.g. a *SUNMatrix* and an *N\_Vector*) since they do not hold any modifiable state information.

## 8.12 The NVECTOR\_SYCL Module

The NVECTOR\_SYCL module is an experimental NVECTOR implementation using the SYCL abstraction layer. At present the only supported SYCL compiler is the DPC++ (Intel oneAPI) compiler. This module allows for SUNDIALS vector kernels to run on Intel GPU devices. The module is intended for users who are already familiar with SYCL and GPU programming.

The vector content layout is as follows:

```
struct _N_VectorContent_Sycl
{
    sunindextype      length;
    sunboolean_t      own_helper;
    SUNMemory         host_data;
    SUNMemory         device_data;
    SUNSyclExecPolicy* stream_exec_policy;
    SUNSyclExecPolicy* reduce_exec_policy;
    SUNMemoryHelper    mem_helper;
    sycl::queue*       queue;
    void*              priv; /* 'private' data */
};

typedef struct _N_VectorContent_Sycl *N_VectorContent_Sycl;
```

The content members are the vector length (size), boolean flags that indicate if the vector owns the execution policies and memory helper objects (i.e., it is in charge of freeing the objects), *SUNMemory* objects for the vector data on the host and device, pointers to execution policies that control how streaming and reduction kernels are launched, a *SUNMemoryHelper* for performing memory operations, the SYCL queue, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with *N\_VNew\_Sycl()*, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the *N\_VMake\_Sycl()* constructor. To use managed (shared) memory, the constructors *N\_VNewManaged\_Sycl()* and *N\_VMakeManaged\_Sycl()* are provided. Additionally, a user-defined *SUNMemoryHelper* for allocating/freeing data can be provided with the constructor *N\_VNewWithMemHelp\_Sycl()*. Details on each of these constructors are provided below.

The header file to include when using this is *nvector\_sycl.h*. The installed module library to link to is *libsundials\_nvecsycl.lib*. The extension *.lib* is typically *.so* for shared libraries *.a* for static libraries.

### 8.12.1 NVECTOR\_SYCL functions

The NVECTOR\_SYCL module implementations of all vector operations listed in §8.2, §8.2.2, §8.2.3, and §8.2.4, except for *N\_VDotProdMulti()*, *N\_VWrmsNormVectorArray()*, *N\_VWrmsNormMaskVectorArray()* as support for

arrays of reduction vectors is not yet supported. These functions will be added to the NVECTOR\_SYCL implementation in the future. The names of vector operations are obtained from those in the aforementioned sections by appending the suffix `_Sycl` (e.g., `N_VDestroy_Sycl`).

Additionally, the NVECTOR\_SYCL module provides the following user-callable constructors for creating a new NVECTOR\_SYCL:

`N_Vector` **N\_VNew\_Sycl**(`sunindextype` `vec_length`, `sycl::queue` `*Q`, `SUNContext` `sunctx`)

This function creates and allocates memory for an NVECTOR\_SYCL. Vector data arrays are allocated on both the host and the device associated with the input queue. All operation are launched in the provided queue.

`N_Vector` **N\_VNewManaged\_Sycl**(`sunindextype` `vec_length`, `sycl::queue` `*Q`, `SUNContext` `sunctx`)

This function creates and allocates memory for a NVECTOR\_SYCL. The vector data array is allocated in managed (shared) memory using the input queue. All operation are launched in the provided queue.

`N_Vector` **N\_VMake\_Sycl**(`sunindextype` `length`, `sunrealtype` `*h_vdata`, `sunrealtype` `*d_vdata`, `sycl::queue` `*Q`, `SUNContext` `sunctx`)

This function creates an NVECTOR\_SYCL with user-supplied host and device data arrays. This function does not allocate memory for data itself. All operation are launched in the provided queue.

`N_Vector` **N\_VMakeManaged\_Sycl**(`sunindextype` `length`, `sunrealtype` `*vdata`, `sycl::queue` `*Q`, `SUNContext` `sunctx`)

This function creates an NVECTOR\_SYCL with a user-supplied managed (shared) data array. This function does not allocate memory for data itself. All operation are launched in the provided queue.

`N_Vector` **N\_VNewWithMemHelp\_Sycl**(`sunindextype` `length`, `sunboolean` `use_managed_mem`, `SUNMemoryHelper` `helper`, `sycl::queue` `*Q`, `SUNContext` `sunctx`)

This function creates an NVECTOR\_SYCL with a user-supplied SUNMemoryHelper for allocating/freeing memory. All operation are launched in the provided queue.

`N_Vector` **N\_VNewEmpty\_Sycl**()

This function creates a new `N_Vector` where the members of the content structure have not been allocated. This utility function is used by the other constructors to create a new vector.

The following user-callable functions are provided for accessing the vector data arrays on the host and device and copying data between the two memory spaces. Note the generic NVECTOR operations `N_VGetArrayPointer()` and `N_VSetArrayPointer()` are mapped to the corresponding `HostArray` functions given below. To ensure memory coherency, a user will need to call the `CopyTo` or `CopyFrom` functions as necessary to transfer data between the host and device, unless managed (shared) memory is used.

`sunrealtype` **N\_VGetHostArrayPointer\_Sycl**(`N_Vector` `v`)

This function returns a pointer to the vector host data array.

`sunrealtype` **N\_VGetDeviceArrayPointer\_Sycl**(`N_Vector` `v`)

This function returns a pointer to the vector device data array.

`void` **N\_VSetHostArrayPointer\_Sycl**(`sunrealtype` `*h_vdata`, `N_Vector` `v`)

This function sets the host array pointer in the vector `v`.

`void` **N\_VSetDeviceArrayPointer\_Sycl**(`sunrealtype` `*d_vdata`, `N_Vector` `v`)

This function sets the device array pointer in the vector `v`.

`void` **N\_VCopyToDevice\_Sycl**(`N_Vector` `v`)

This function copies host vector data to the device.

`void` **N\_VCopyFromDevice\_Sycl**(`N_Vector` `v`)

This function copies vector data from the device to the host.



sunboolean\_t **N\_VIsManagedMemory\_Sycl**(N\_Vector v)

This function returns `SUNTRUE` if the vector data is allocated as managed (shared) memory otherwise it returns `SUNFALSE`.

The following user-callable function is provided to set the execution policies for how SYCL kernels are launched on a device.

SUNErrCode **N\_VSetKernelExecPolicy\_Sycl**(N\_Vector v, *SUNSyclExecPolicy* \*stream\_exec\_policy, *SUNSyclExecPolicy* \*reduce\_exec\_policy)

This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction kernels. By default the vector is setup to use the *SUNSyclThreadDirectExecPolicy()* and *SUNSyclBlockReduceExecPolicy()*. See §8.12.2 below for more information about the *SUNSyclExecPolicy* class.

The input execution policies are cloned and, as such, may be freed after being attached to the desired vectors. A `NULL` input policy will reset the execution policy to the default setting.

#### Note

All vectors used in a single instance of a SUNDIALS package must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

The following user-callable functions are provided to print the host vector data array. Unless managed memory is used, a user may need to call *N\_VCopyFromDevice\_Sycl()* to ensure consistency between the host and device array.

void **N\_VPrint\_Sycl**(N\_Vector v)

This function prints the host data array to `stdout`.

void **N\_VPrintFile\_Sycl**(N\_Vector v, FILE \*outfile)

This function prints the host data array to `outfile`.

By default all fused and vector array operations are disabled in the `NVECTOR_SYCL` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with one of the above constructors, enable/disable the desired operations on that vector with the functions below, and then use this vector in conjunction with *N\_VClone()* to create any additional vectors. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created by any of the constructors above will have the default settings for the `NVECTOR_SYCL` module.

SUNErrCode **N\_VEnableFusedOps\_Sycl**(N\_Vector v, sunboolean\_t tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the SYCL vector. The return value is a *SUNErrCode*.

SUNErrCode **N\_VEnableLinearCombination\_Sycl**(N\_Vector v, sunboolean\_t tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the SYCL vector. The return value is a *SUNErrCode*.

SUNErrCode **N\_VEnableScaleAddMulti\_Sycl**(N\_Vector v, sunboolean\_t tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the SYCL vector. The return value is a *SUNErrCode*.

SUNErrCode **N\_VEnableLinearSumVectorArray\_Sycl**(N\_Vector v, sunboolean\_t tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the SYCL vector. The return value is a *SUNErrCode*.

SUNErrCode **N\_VEnableScaleVectorArray\_Sycl**(N\_Vector v, sunbooleantype tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the SYCL vector. The return value is a [SUNErrCode](#).

SUNErrCode **N\_VEnableConstVectorArray\_Sycl**(N\_Vector v, sunbooleantype tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the SYCL vector. The return value is a [SUNErrCode](#).

SUNErrCode **N\_VEnableScaleAddMultiVectorArray\_Sycl**(N\_Vector v, sunbooleantype tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the SYCL vector. The return value is a [SUNErrCode](#).

SUNErrCode **N\_VEnableLinearCombinationVectorArray\_Sycl**(N\_Vector v, sunbooleantype tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the SYCL vector. The return value is a [SUNErrCode](#).

#### Notes

- When there is a need to access components of an NVECTOR\_SYCL, v, it is recommended to use [N\\_VGetDeviceArrayPointer\(\)](#) to access the device array or [N\\_VGetArrayPointer\(\)](#) for the host array. When using managed (shared) memory, either function may be used. To ensure memory coherency, a user may need to call the `CopyTo` or `CopyFrom` functions as necessary to transfer data between the host and device, unless managed (shared) memory is used.
- To maximize efficiency, vector operations in the NVECTOR\_SYCL 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.

### 8.12.2 The SUNSyclExecPolicy Class

In order to provide maximum flexibility to users, the SYCL kernel execution parameters used by kernels within SUNDIALS are defined by objects of the `sundials::sycl::ExecPolicy` abstract class type (this class can be accessed in the global namespace as `SUNSyclExecPolicy`). Thus, users may provide custom execution policies that fit the needs of their problem. The `SUNSyclExecPolicy` class is defined as

```
typedef sundials::sycl::ExecPolicy SUNSyclExecPolicy
```

where the `sundials::sycl::ExecPolicy` class is defined in the header file `sundials_sycl_policies.hpp`, as follows:

```
class sundials::sycl::ExecPolicy
```

```
    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0)
```

```
    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0)
```

```
    virtual ExecPolicy *clone() const
```

```
    virtual ~ExecPolicy()
```

For consistency the function names and behavior mirror the execution policies for the CUDA and HIP vectors. In the SYCL case the `blockSize` is the local work-group range in a one-dimensional `nd_range` (threads per group). The `gridSize` is the number of local work groups so the global work-group range in a one-dimensional `nd_range` is `blockSize * gridSize` (total number of threads). All vector kernels are written with a many-to-one mapping where work units (vector elements) are mapped in a round-robin manner across the global range. As such, the `blockSize` and `gridSize` can be set to any positive value.



To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::sycl::ThreadDirectExecPolicy` (aka in the global namespace as `SUNSYCLThreadDirectExecPolicy`) class is a good example of what a custom execution policy may look like:

```
class ThreadDirectExecPolicy : public ExecPolicy
{
public:
    ThreadDirectExecPolicy(const size_t blockDim)
        : blockDim_(blockDim)
    {}

    ThreadDirectExecPolicy(const ThreadDirectExecPolicy& ex)
        : blockDim_(ex.blockDim_)
    {}

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
    {
        return (numWorkUnits + blockSize() - 1) / blockSize();
    }

    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
    {
        return blockDim_;
    }

    virtual ExecPolicy* clone() const
    {
        return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
    }

private:
    const size_t blockDim_;
};
```

SUNDIALS provides the following execution policies:

#### **SUNSYCLThreadDirectExecPolicy**(const size\_t blockDim)

Is for kernels performing streaming operations and maps each work unit (vector element) to a work-item (thread). Based on the local work-group range (number of threads per group, `blockSize`) the number of local work-groups (`gridSize`) is computed so there are enough work-items in the global work-group range (total number of threads, `blockSize * gridSize`) for one work unit per work-item (thread).

#### **SUNSYCLGridStrideExecPolicy**(const size\_t blockDim, const size\_t gridDim)

Is for kernels performing streaming operations and maps each work unit (vector element) to a work-item (thread) in a round-robin manner so the local work-group range (number of threads per group, `blockSize`) and the number of local work-groups (`gridSize`) can be set to any positive value. In this case the global work-group range (total number of threads, `blockSize * gridSize`) may be less than the number of work units (vector elements).

#### **SUNSYCLBlockReduceExecPolicy**(const size\_t blockDim)

Is for kernels performing a reduction, the local work-group range (number of threads per group, `blockSize`) and the number of local work-groups (`gridSize`) can be set to any positive value or the `gridSize` may be set to 0 in which case the global range is chosen so that there are enough threads

for at most two work units per work-item.

By default the NVECTOR\_SYCL module uses the `SUNSyclThreadDirectExecPolicy` and `SUNSyclBlockReduceExecPolicy` where the default `blockDim` is determined by querying the device for the `max_work_group_size`. User may specify different policies by constructing a new `SyclExecPolicy` and attaching it with `N_VSetKernelExecPolicy_Sycl()`. For example, a policy that uses 128 work-items (threads) per group can be created and attached like so:

```
N_Vector v = N_VNew_Sycl(length, SUNContext sunctx);
SUNSyclThreadDirectExecPolicy thread_direct(128);
SUNSyclBlockReduceExecPolicy block_reduce(128);
flag = N_VSetKernelExecPolicy_Sycl(v, &thread_direct, &block_reduce);
```

These default policy objects can be reused for multiple SUNDIALS data structures (e.g. a `SUNMatrix` and an `N_Vector`) since they do not hold any modifiable state information.

## 8.13 The NVECTOR\_RAJA Module

The NVECTOR\_RAJA module is an experimental NVECTOR implementation using the `RAJA` hardware abstraction layer. In this implementation, RAJA allows for SUNDIALS vector kernels to run on AMD, NVIDIA, or Intel GPU devices. The module is intended for users who are already familiar with RAJA and GPU programming. Building this vector module requires a C++11 compliant compiler and either the NVIDIA CUDA programming environment, the AMD ROCm HIP programming environment, or a compiler that supports the SYCL abstraction layer. When using the AMD ROCm HIP environment, the HIP-clang compiler must be utilized. Users can select which backend to compile with by setting the `SUNDIALS_RAJA_BACKENDS` CMake variable to either CUDA, HIP, or SYCL. Besides the CUDA, HIP, and SYCL backends, RAJA has other backends such as serial, OpenMP, and OpenACC. These backends are not used in this SUNDIALS release.

The vector content layout is as follows:

```
struct _N_VectorContent_Raja
{
    sunindextype length;
    sunbooleantype own_data;
    sunrealtype* host_data;
    sunrealtype* device_data;
    void* priv; /* 'private' data */
};
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the data (i.e., it is in charge of freeing the data), pointers to vector data on the host and the device, and a private data structure which holds the memory management type, which should not be accessed directly.

When instantiated with `N_VNew_Raja()`, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the `N_VMake_Raja()` constructor. To use managed memory, the constructors `N_VNewManaged_Raja()` and `N_VMakeManaged_Raja()` are provided. Details on each of these constructors are provided below.

The header file to include when using this is `nvector_raj.h`. The installed module library to link to is `libsundials_nveccudaraja.lib` when using the CUDA backend, `libsundials_nvechipraja.lib` when using the HIP backend, and `libsundials_nvecsyclraja.lib` when using the SYCL backend. The extension `.lib` is typically `.so` for shared libraries `.a` for static libraries.

### 8.13.1 NVECTOR\_RAJA functions

Unlike other native SUNDIALS vector types, the NVECTOR\_RAJA module does not provide macros to access its member variables. Instead, user should use the accessor functions:

*sunrealtype* \***N\_VGetHostArrayPointer\_Raja**(*N\_Vector* v)

This function returns pointer to the vector data on the host.

*sunrealtype* \***N\_VGetDeviceArrayPointer\_Raja**(*N\_Vector* v)

This function returns pointer to the vector data on the device.

*sunbooleantype* **N\_VIsManagedMemory\_Raja**(*N\_Vector* v)

This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The NVECTOR\_RAJA module defines the implementations of all vector operations listed in §8.2, §8.2.2, §8.2.3, and §8.2.4, except for *N\_VDotProdMulti()*, *N\_VWrmsNormVectorArray()*, and *N\_VWrmsNormMaskVectorArray()* as support for arrays of reduction vectors is not yet supported in RAJA. These functions will be added to the NVECTOR\_RAJA implementation in the future. Additionally, the operations *N\_VGetArrayPointer()* and *N\_VSetArrayPointer()* are not implemented by the RAJA vector. As such, this vector cannot be used with SUNDIALS 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 [62].

The names of vector operations are obtained from those in §8.2, §8.2.2, §8.2.3, and §8.2.4 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, *SUNContext* sunctx)

This function creates and allocates memory for a RAJA *N\_Vector*. The memory is allocated on both the host and the device. Its only argument is the vector length.

*N\_Vector* **N\_VNewManaged\_Raja**(*sunindextype* vec\_length, *SUNContext* sunctx)

This function creates and allocates memory for a RAJA *N\_Vector*. The vector data array is allocated in managed memory.

*N\_Vector* **N\_VMake\_Raja**(*sunindextype* length, *sunrealtype* \*h\_data, *sunrealtype* \*v\_data, *SUNContext* sunctx)

This function creates an NVECTOR\_RAJA with user-supplied host and device data arrays. This function does not allocate memory for data itself.

*N\_Vector* **N\_VMakeManaged\_Raja**(*sunindextype* length, *sunrealtype* \*vdata, *SUNContext* sunctx)

This function creates an NVECTOR\_RAJA with a user-supplied managed memory data array. This function does not allocate memory for data itself.

*N\_Vector* **N\_VNewWithMemHelp\_Raja**(*sunindextype* length, *sunbooleantype* use\_managed\_mem, *SUNMemoryHelper* helper, *SUNContext* sunctx)

This function creates an NVECTOR\_RAJA with a user-supplied *SUNMemoryHelper* for allocating/freeing memory.

*N\_Vector* **N\_VNewEmpty\_Raja**()

This function creates a new *N\_Vector* where the members of the content structure have not been allocated. This utility function is used by the other constructors to create a new vector.

void **N\_VCopyToDevice\_Raja**(*N\_Vector* v)

This function copies host vector data to the device.

void **N\_VCopyFromDevice\_Raja**(*N\_Vector* v)

This function copies vector data from the device to the host.

void **N\_VPrint\_Raja**(*N\_Vector* v)

This function prints the content of a RAJA vector to `stdout`.

void **N\_VPrintFile\_Raja**(*N\_Vector* v, FILE \*outfile)

This function prints the content of a RAJA vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR\_RAJA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Raja()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Raja()` will have the default settings for the NVECTOR\_RAJA module.

*SUNErrCode* **N\_VEnableFusedOps\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the RAJA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the RAJA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the RAJA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the RAJA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the RAJA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the RAJA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the RAJA vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_Raja**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the RAJA vector. The return value is a *SUNErrCode*.

## Notes

- When there is a need to access components of an NVECTOR\_RAJA vector, it is recommended to use functions `N_VGetDeviceArrayPointer_Raja()` or `N_VGetHostArrayPointer_Raja()`. However, when using managed memory, the function `N_VGetArrayPointer()` may also be used.
- To maximize efficiency, vector operations in the NVECTOR\_RAJA implementation that have more than one *N\_Vector* argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with *N\_Vector* arguments that were all created with the same internal representations.

## 8.14 The NVECTOR\_KOKKOS Module

Added in version 6.4.0.

The NVECTOR\_KOKKOS *N\_Vector* implementation provides a vector data structure using Kokkos [39, 119] to support a variety of backends including serial, OpenMP, CUDA, HIP, and SYCL. Since Kokkos is a modern C++ library, the module is also written in modern C++ (it requires C++14) as a header only library. To utilize this *N\_Vector* users will need to include `nvector/nvector_kokkos.hpp`. More instructions on building SUNDIALS with Kokkos enabled are given in §17.3.23. For instructions on building and using Kokkos, refer to the [Kokkos](#) documentation.

### 8.14.1 Using NVECTOR\_KOKKOS

The NVECTOR\_KOKKOS module is defined by the *Vector* templated class in the `sundials::kokkos` namespace:

```
template<class ExecutionSpace = Kokkos::DefaultExecutionSpace,
        class MemorySpace = typename ExecutionSpace::memory_space>
class Vector : public sundials::impl::BaseNVector,
               public sundials::ConvertibleTo<N_Vector>
```

To use the NVECTOR\_KOKKOS module, we construct an instance of the *Vector* class e.g.,

```
// Vector with extent length using the default execution space
sundials::kokkos::Vector<> x{length, sunctx};

// Vector with extent length using the Cuda execution space
sundials::kokkos::Vector<Kokkos::Cuda> x{length, sunctx};

// Vector based on an existing Kokkos::View
Kokkos::View<> view{"a view", length};
sundials::kokkos::Vector<> x{view, sunctx};

// Vector based on an existing Kokkos::View for device and host
Kokkos::View<Kokkos::Cuda> device_view{"a view", length};
Kokkos::View<Kokkos::HostMirror> host_view{Kokkos::create_mirror_view(device_view)};
sundials::kokkos::Vector<> x{device_view, host_view, sunctx};
```

Instances of the *Vector* class are implicitly or explicitly (using the *get()* method) convertible to a *N\_Vector* e.g.,

```
sundials::kokkos::Vector<> x{length, sunctx};
N_Vector x2 = x;           // implicit conversion to N_Vector
N_Vector x3 = x.get();     // explicit conversion to N_Vector
```

No further interaction with a *Vector* is required from this point, and it is possible to use the *N\_Vector* API to operate on *x2* or *x3*.

#### Warning

*N\_VDestroy()* should never be called on a *N\_Vector* that was created via conversion from a `sundials::kokkos::Vector`. Doing so may result in a double free.

The underlying *Vector* can be extracted from a *N\_Vector* using *GetVec()* e.g.,

```
auto x_vec = GetVec<>(x3);
```

### 8.14.2 NVECTOR\_KOKKOS API

In this section we list the public API of the `sundials::kokkos::Vector` class.

```
template<class ExecutionSpace = Kokkos::DefaultExecutionSpace, class MemorySpace = class  
ExecutionSpace::memory_space>
```

```
class Vector : public sundials::impl::BaseNVector, public sundials::ConvertibleTo<N_Vector>
```

```
using view_type = Kokkos::View<sunrealtype*, MemorySpace>;
```

```
using size_type = typename view_type::size_type;
```

```
using host_view_type = typename view_type::HostMirror;
```

```
using memory_space = MemorySpace;
```

```
using exec_space = typename MemorySpace::execution_space;
```

```
using range_policy = Kokkos::RangePolicy<exec_space>;
```

```
Vector() = default
```

Default constructor – the vector must be copied or moved to.

```
Vector(size_type length, SUNContext sunctx)
```

Constructs a single `Vector` which is based on a 1D `Kokkos::View` with the `ExecutionSpace` and `MemorySpace` provided as template arguments.

#### Parameters

- **length** – length of the vector (i.e., the extent of the `View`)
- **sunctx** – the SUNDIALS simulation context object (*SUNContext*)

```
Vector(view_type view, SUNContext sunctx)
```

Constructs a single `Vector` from an existing `Kokkos::View`. The `View` `ExecutionSpace` and `MemorySpace` must match the `ExecutionSpace` and `MemorySpace` provided as template arguments.

#### Parameters

- **view** – A 1D `Kokkos::View`
- **sunctx** – the SUNDIALS simulation context object (*SUNContext*)

```
Vector(view_type view, host_view_type host_view, SUNContext sunctx)
```

Constructs a single `Vector` from an existing `Kokkos::View` for the device and the host. The `ExecutionSpace` and `MemorySpace` of the device `View` must match the `ExecutionSpace` and `MemorySpace` provided as template arguments.

#### Parameters

- **view** – A 1D `Kokkos::View` for the device
- **host\_view** – A 1D `Kokkos::View` that is a `Kokkos::HostMirror` for the device view
- **sunctx** – the SUNDIALS simulation context object (*SUNContext*)

```
Vector(Vector &&that_vector) noexcept
```

Move constructor.

**Vector**(const *Vector* &that\_vector)

Copy constructor. This creates a clone of the Vector, i.e., it creates a new Vector with the same properties, such as length, but it does not copy the data.

*Vector* &**operator**=(*Vector* &&rhs) noexcept

Move assignment.

*Vector* &**operator**=(const *Vector* &rhs)

Copy assignment. This creates a clone of the Vector, i.e., it creates a new Vector with the same properties, such as length, but it does not copy the data.

virtual ~**Vector**() = default;

Default destructor.

*size\_type* **Length**()

Get the vector length i.e., extent(0).

*view\_type* **View**()

Get the underlying Kokkos:View for the device.

*host\_view\_type* **HostView**()

Get the underlying Kokkos:View for the host.

**operator N\_Vector**() override

Implicit conversion to a *N\_Vector*.

**operator N\_Vector**() const override

Implicit conversion to a *N\_Vector*.

*N\_Vector* **get**() override

Explicit conversion to a *N\_Vector*.

Added in version 7.6.0: Replaces the Convert method which was deprecated.

*N\_Vector* **get**() const override

Explicit conversion to a *N\_Vector*.

Added in version 7.6.0: Replaces the Convert method which was deprecated.

template<class **VectorType**>

inline *VectorType* \***GetVec**(*N\_Vector* v)

Get the *Vector* wrapped by a *N\_Vector*.

void **CopyToDevice**(*N\_Vector* v)

Copy the data from the host view to the device view with Kokkos::deep\_copy.

void **CopyFromDevice**(*N\_Vector* v)

Copy the data to the host view from the device view with Kokkos::deep\_copy.

template<class **VectorType**>

void **CopyToDevice**(*VectorType* &v)

Copy the data from the host view to the device view with Kokkos::deep\_copy.

template<class **VectorType**>

void **CopyFromDevice**(*VectorType* &v)

Copy the data to the host view from the device view with Kokkos::deep\_copy.



## 8.15 The NVECTOR\_OPENMPDEV Module

In situations where a user has access to a device such as a GPU for offloading computation, SUNDIALS provides an NVECTOR implementation using OpenMP device offloading, called NVECTOR\_OPENMPDEV.

The NVECTOR\_OPENMPDEV implementation defines the *content* field of the `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array on the host, a pointer to the beginning of a contiguous data array on the device, and a boolean flag `own_data` which specifies the ownership of host and device data arrays.

```
struct _N_VectorContent_OpenMPDEV
{
    sunindextype length;
    sunbooleantype own_data;
    sunrealtype    *host_data;
    sunrealtype    *dev_data;
};
```

The header file to include when using this module is `nvector_openmpdev.h`. The installed module library to link to is `libsundials_nvecopenmpdev.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

### 8.15.1 NVECTOR\_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVECTOR\_OPENMPDEV vector.

#### NV\_CONTENT\_OMPDEV(v)

This macro gives access to the contents of the NVECTOR\_OPENMPDEV `N_Vector` `v`.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the NVECTOR\_OPENMPDEV content structure.

Implementation:

```
#define NV_CONTENT_OMPDEV(v) ( (N_VectorContent_OpenMPDEV)(v->content) )
```

#### NV\_OWN\_DATA\_OMPDEV(v)

Access the `own_data` component of the OpenMPDEV `N_Vector` `v`.

The assignment `v_data = NV_DATA_HOST_OMPDEV(v)` sets `v_data` to be a pointer to the first component of the data on the host for the `N_Vector` `v`.

Implementation:

```
#define NV_OWN_DATA_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->own_data )
```

#### NV\_DATA\_HOST\_OMPDEV(v)

The assignment `NV_DATA_HOST_OMPDEV(v) = v_data` sets the host component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_HOST_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->host_data )
```



**NV\_DATA\_DEV\_OMPDEV(v)**

The assignment `v_dev_data = NV_DATA_DEV_OMPDEV(v)` sets `v_dev_data` to be a pointer to the first component of the data on the device for the `N_Vector v`. The assignment `NV_DATA_DEV_OMPDEV(v) = v_dev_data` sets the device component array of `v` to be `v_dev_data` by storing the pointer `v_dev_data`.

Implementation:

```
#define NV_DATA_DEV_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->dev_data )
```

**NV\_LENGTH\_OMPDEV(V)**

Access the *length* component of the OpenMPDEV `N_Vector v`.

The assignment `v_len = NV_LENGTH_OMPDEV(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMPDEV(v) = len_v` sets the length of `v` to be `len_v`.

```
#define NV_LENGTH_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->length )
```

**8.15.2 NVECTOR\_OPENMPDEV functions**

The `NVECTOR_OPENMPDEV` module defines OpenMP device offloading implementations of all vector operations listed in §8.2, §8.2.2, §8.2.3, and §8.2.4, except for `N_VSetArrayPointer()`. As such, this vector cannot be used with the SUNDIALS direct solvers and preconditioners. It also provides methods for copying from the host to the device and vice versa.

The names of the vector operations are obtained from those in §8.2, §8.2.2, §8.2.3, and §8.2.4 by appending the suffix `_OpenMPDEV` (e.g. `N_VDestroy_OpenMPDEV`). The module `NVECTOR_OPENMPDEV` provides the following additional user-callable routines:

*N\_Vector* **N\_VNew\_OpenMPDEV**(*sunindextype* vec\_length, *SUNContext* sunctx)

This function creates and allocates memory for an `NVECTOR_OPENMPDEV N_Vector`.

*N\_Vector* **N\_VNewEmpty\_OpenMPDEV**(*sunindextype* vec\_length, *SUNContext* sunctx)

This function creates a new `NVECTOR_OPENMPDEV N_Vector` with an empty (NULL) data array.

*N\_Vector* **N\_VMake\_OpenMPDEV**(*sunindextype* vec\_length, *sunrealtype* \*h\_vdata, *sunrealtype* \*d\_vdata, *SUNContext* sunctx)

This function creates an `NVECTOR_OPENMPDEV` vector with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself.

*sunrealtype* \***N\_VGetHostArrayPointer\_OpenMPDEV**(*N\_Vector* v)

This function returns a pointer to the host data array.

*sunrealtype* \***N\_VGetDeviceArrayPointer\_OpenMPDEV**(*N\_Vector* v)

This function returns a pointer to the device data array.

void **N\_VPrint\_OpenMPDEV**(*N\_Vector* v)

This function prints the content of an `NVECTOR_OPENMPDEV` vector to `stdout`.

void **N\_VPrintFile\_OpenMPDEV**(*N\_Vector* v, FILE \*outfile)

This function prints the content of an `NVECTOR_OPENMPDEV` vector to `outfile`.

void **N\_VCopyToDevice\_OpenMPDEV**(*N\_Vector* v)

This function copies the content of an `NVECTOR_OPENMPDEV` vector's host data array to the device data array.

void **N\_VCopyFromDevice\_OpenMPDEV**(*N\_Vector* v)

This function copies the content of an NVECTOR\_OPENMPDEV vector's device data array to the host data array.

By default all fused and vector array operations are disabled in the NVECTOR\_OPENMPDEV module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with **N\_VNew\_OpenMPDEV**, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using **N\_VClone**. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with **N\_VNew\_OpenMPDEV** will have the default settings for the NVECTOR\_OPENMPDEV module.

*SUNErrCode* **N\_VEnableFusedOps\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMultiVectorArray\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombinationVectorArray\_OpenMPDEV**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is a *SUNErrCode*.

## Notes

- When looping over the components of an `N_Vector v`, it is most efficient to first obtain the component array via `h_data = N_VGetArrayPointer(v)` for the host array or `v_data = N_VGetDeviceArrayPointer(v)` for the device array, or equivalently to use the macros `h_data = NV_DATA_HOST_OMPDEV(v)` for the host array or `v_data = NV_DATA_DEV_OMPDEV(v)` for the device array, and then access `h_data[i]` or `v_data[i]` within the loop.
- When accessing individual components of an `N_Vector v` on the host remember to first copy the array back from the device with `N_VCopyFromDevice_OpenMPDEV(v)` to ensure the array is up to date.
- `N_VNewEmpty_OpenMPDEV()` and `N_VMake_OpenMPDEV()` set the field `own_data` to `SUNFALSE`. The implementation of `N_VDestroy()` will not attempt to free the pointer data for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the data pointers.
- To maximize efficiency, vector operations in the `NVECTOR_OPENMPDEV` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same length.

## 8.16 The NVECTOR\_TRILINOS Module

The `NVECTOR_TRILINOS` module is an `NVECTOR` wrapper around the [Trilinos](#) Tpetra vector. The interface to Tpetra is implemented in the `sundials::trilinos::nvector_tpetra::TpetraVectorInterface` class. This class simply stores a reference counting pointer to a Tpetra vector and inherits from an empty structure

```
struct _N_VectorContent_Trilinos {};
```

to interface the C++ class with the `NVECTOR` C code. A pointer to an instance of this class is kept in the `content` field of the `N_Vector` object, to ensure that the Tpetra vector is not deleted for as long as the `N_Vector` object exists.

The Tpetra vector type in the `sundials::trilinos::nvector_tpetra::TpetraVectorInterface` class is defined as:

```
typedef Tpetra::Vector<sunrealtype, int, sunindextype> vector_type;
```

The Tpetra vector will use the SUNDIALS-specified `sunrealtype` as its scalar type, `int` as the local ordinal type, and `sunindextype` as the global ordinal type. This type definition will use Tpetra's default node type. Available Kokkos node types as of the Trilinos 12.14 release are serial (single thread), OpenMP, Pthread, and CUDA. The default node type is selected when building the Kokkos package. For example, the Tpetra vector will use a CUDA node if Tpetra was built with CUDA support and the CUDA node was selected as the default when Tpetra was built.

The header file to include when using this module is `nvector_trilinos.h`. The installed module library to link to is `libsundials_nvectrilinos.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

### 8.16.1 NVECTOR\_TRILINOS functions

The `NVECTOR_TRILINOS` module defines implementations of all vector operations listed in §8.2, §8.2.2, §8.2.3, and §8.2.4, except for `N_VGetArrayPointer()` and `N_VSetArrayPointer()`. As such, this vector cannot be used with the SUNDIALS direct solvers and preconditioners. When access to raw vector data is needed, it is recommended to extract the Trilinos Tpetra vector first, and then use Tpetra vector methods to access the data. Usage examples of `NVECTOR_TRILINOS` are provided in example programs for IDA.

The names of vector operations are obtained from those in §8.2 by appending the suffix `_Trilinos` (e.g. `N_VDestroy_Trilinos`). Vector operations call existing `Tpetra::Vector` methods when available. Vector operations specific to SUNDIALS are implemented as standalone functions in the namespace `sundials::trilinos::nvector_tpetra::TpetraVector`, located in the file `SundialsTpetraVectorKernels.hpp`. The module `NVECTOR_TRILINOS` provides the following additional user-callable routines:

Teuchos::RCP<*vector\_type*> **N\_VGetVector\_Trilinos**(N\_Vector v)

This C++ function takes an N\_Vector as the argument and returns a reference counting pointer to the underlying Tpetra vector. This is a standalone function defined in the global namespace.

N\_Vector **N\_VMake\_Trilinos**(Teuchos::RCP<*vector\_type*> v)

This C++ function creates and allocates memory for an NVECTOR\_TRILINOS wrapper around a user-provided Tpetra vector. This is a standalone function defined in the global namespace.

#### Notes

- The template parameter `vector_type` should be set as:

```
typedef sundials::trilinos::nvector_tpetra::TpetraVectorInterface::vector_type vector_type
```

This will ensure that data types used in Tpetra vector match those in SUNDIALS.

- When there is a need to access components of an N\_Vector\_Trilinos v, it is recommended to extract the Trilinos vector object via `x_vec = N_VGetVector_Trilinos(v)` and then access components using the appropriate Trilinos functions.
- The function `N_VDestroy_Trilinos` only deletes the N\_Vector wrapper. The underlying Tpetra vector object will exist for as long as there is at least one reference to it.

## 8.17 The NVECTOR\_MANYVECTOR Module

The NVECTOR\_MANYVECTOR module is designed to facilitate problems with an inherent data partitioning within a computational node for the solution vector. These data partitions are entirely user-defined, through construction of distinct NVECTOR modules for each component, that are then combined together to form the NVECTOR\_MANYVECTOR. Two potential use cases for this flexibility include:

- Heterogeneous computational architectures:* for data partitioning between different computing resources on a node, architecture-specific subvectors may be created for each partition. For example, a user could create one GPU-accelerated component based on [NVECTOR\\_CUDA](#), and another CPU threaded component based on [NVECTOR\\_OPENMP](#).
- Structure of arrays (SOA) data layouts:* for problems that require separate subvectors for each solution component. For example, in an incompressible Navier-Stokes simulation, separate subvectors may be used for velocities and pressure, which are combined together into a single NVECTOR\_MANYVECTOR for the overall “solution”.

The above use cases are neither exhaustive nor mutually exclusive, and the NVECTOR\_MANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR\_MANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum “standard” set of operations in §8.2.1. Additionally, NVECTOR\_MANYVECTOR sets no limit on the number of subvectors that may be attached (aside from the limitations of using `sunindextype` for indexing, and standard per-node memory limitations). However, while this ostensibly supports subvectors with one entry each (i.e., one subvector for each solution entry), we anticipate that this extreme situation will hinder performance due to non-stride-one memory accesses and increased function call overhead. We therefore recommend a relatively coarse partitioning of the problem, although actual performance will likely be problem-dependent.

As a final note, in the coming years we plan to introduce additional algebraic solvers and time integration modules that will leverage the problem partitioning enabled by NVECTOR\_MANYVECTOR. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side function, DAE or nonlinear solver residual function, preconditioners, or custom [SUNLinearSolver](#) or [SUNNonlinearSolver](#) modules.

### 8.17.1 NVECTOR\_MANYVECTOR structure

The NVECTOR\_MANYVECTOR implementation defines the *content* field of `N_Vector` to be a structure containing the number of subvectors comprising the ManyVector, the global length of the ManyVector (including all subvectors), a pointer to the beginning of the array of subvectors, and a boolean flag `own_data` indicating ownership of the subvectors that populate `subvec_array`.

```
struct _N_VectorContent_ManyVector {
    sunindextype  num_subvectors; /* number of vectors attached */
    sunindextype  global_length; /* overall manyvector length */
    N_Vector*     subvec_array;   /* pointer to N_Vector array */
    sunbooleantype own_data;      /* flag indicating data ownership */
};
```

The header file to include when using this module is `nvector_manyvector.h`. The installed module library to link against is `libsundials_nvecmanyvector.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

### 8.17.2 NVECTOR\_MANYVECTOR functions

The NVECTOR\_MANYVECTOR module implements all vector operations listed in §8.2 except for `N_VGetArrayPointer()`, `N_VSetArrayPointer()`, `N_VScaleAddMultiVectorArray()`, and `N_VLinearCombinationVectorArray()`. As such, this vector cannot be used with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR\_MANYVECTOR module provides functions to access subvectors, whose data may in turn be accessed according to their NVECTOR implementations.

The names of vector operations are obtained from those in §8.2 by appending the suffix `_ManyVector` (e.g. `N_VDestroy_ManyVector`). The module NVECTOR\_MANYVECTOR provides the following additional user-callable routines:

*N\_Vector* **N\_VNew\_ManyVector**(*sunindextype* num\_subvectors, *N\_Vector*\* vec\_array, *SUNContext* sunctx)

This function creates a ManyVector from a set of existing NVECTOR objects.

This routine will copy all `N_Vector` pointers from the input `vec_array`, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the ManyVector that contains them.

Upon successful completion, the new ManyVector is returned; otherwise this routine returns NULL (e.g., a memory allocation failure occurred).

Users of the Fortran 2003 interface to this function will first need to use the generic `N_Vector` utility functions `N_VNewVectorArray()`, and `N_VSetVecAtIndexVectorArray()` to create the `N_Vector*` argument. This is further explained in §20.3.5, and the functions are documented in §8.1.1.

*N\_Vector* **N\_VGetSubvector\_ManyVector**(*N\_Vector* v, *sunindextype* vec\_num)

This function returns the `vec_num` subvector from the NVECTOR array.

*sunindextype* **N\_VGetSubvectorLocalLength\_ManyVector**(*N\_Vector* v, *sunindextype* vec\_num)

This function returns the local length of the `vec_num` subvector from the NVECTOR array.

Usage:

```
local_length = N_VGetSubvectorLocalLength_ManyVector(v, 0);
```

*sunrealtype*\* **N\_VGetSubvectorArrayPointer\_ManyVector**(*N\_Vector* v, *sunindextype* vec\_num)

This function returns the data array pointer for the `vec_num` subvector from the NVECTOR array.



If the input *vec\_num* is invalid, or if the subvector does not support the `N_VGetArrayPointer` operation, then `NULL` is returned.

***SUNErrCode* N\_VSetSubvectorArrayPointer\_ManyVector(*sunrealtype* \*v\_data, *N\_Vector* v, *sunindextype* vec\_num)**

This function sets the data array pointer for the *vec\_num* subvector from the `NVECTOR` array.

The function returns a *SUNErrCode*.

***sunindextype* N\_VGetNumSubvectors\_ManyVector(*N\_Vector* v)**

This function returns the overall number of subvectors in the `ManyVector` object.

By default all fused and vector array operations are disabled in the `NVECTOR_MANYVECTOR` module, except for `N_VWrmsNormVectorArray()` and `N_VWrmsNormMaskVectorArray()`, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_ManyVector()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with `N_VNew_ManyVector()` will have the default settings for the `NVECTOR_MANYVECTOR` module. We note that these routines *do not* call the corresponding routines on subvectors, so those should be set up as desired *before* attaching them to the `ManyVector` in `N_VNew_ManyVector()`.

***SUNErrCode* N\_VEnableFusedOps\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableLinearCombination\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableScaleAddMulti\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableDotProdMulti\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableLinearSumVectorArray\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableScaleVectorArray\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableConstVectorArray\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableWrmsNormVectorArray\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the WRMS norm operation for vector arrays in the `manyvector` vector. The return value is a *SUNErrCode*.

***SUNErrCode* N\_VEnableWrmsNormMaskVectorArray\_ManyVector(*N\_Vector* v, *sunbooleantype* tf)**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the masked WRMS norm operation for vector arrays in the `manyvector` vector. The return value is a *SUNErrCode*.

**Notes**

- `N_VNew_ManyVector()` sets the field `own_data = SUNFALSE`. The `ManyVector` implementation of `N_VDestroy()` will not attempt to call `N_VDestroy()` on any subvectors contained in the subvector array for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the subvectors.
- To maximize efficiency, arithmetic vector operations in the `NVECTOR_MANYVECTOR` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same subvector representations.

**8.18 The NVECTOR\_MPIMANYVECTOR Module**

The `NVECTOR_MPIMANYVECTOR` module is designed to facilitate problems with an inherent data partitioning for the solution vector, and when using distributed-memory parallel architectures. As such, this implementation supports all use cases allowed by the MPI-unaware `NVECTOR_MANYVECTOR` implementation, as well as partitioning data between nodes in a parallel environment. These data partitions are entirely user-defined, through construction of distinct `NVECTOR` modules for each component, that are then combined together to form the `NVECTOR_MPIMANYVECTOR`. Three potential use cases for this module include:

- Heterogeneous computational architectures (single-node or multi-node):* for data partitioning between different computing resources on a node, architecture-specific subvectors may be created for each partition. For example, a user could create one MPI-parallel component based on `NVECTOR_PARALLEL`, another GPU-accelerated component based on `NVECTOR_CUDA`.
- Process-based multiphysics decompositions (multi-node):* for computations that combine separate MPI-based simulations together, each subvector may reside on a different MPI communicator, and the `MPIManyVector` combines these via an MPI *intercommunicator* that connects these distinct simulations together.
- Structure of arrays (SOA) data layouts (single-node or multi-node):* for problems that require separate subvectors for each solution component. For example, in an incompressible Navier-Stokes simulation, separate subvectors may be used for velocities and pressure, which are combined together into a single `MPIManyVector` for the overall "solution".

The above use cases are neither exhaustive nor mutually exclusive, and the `NVECTOR_MPIMANYVECTOR` implementation should support arbitrary combinations of these cases.

The `NVECTOR_MPIMANYVECTOR` implementation is designed to work with any `NVECTOR` subvectors that implement the minimum "standard" set of operations in §8.2.1, however significant performance benefits may be obtained when subvectors additionally implement the optional local reduction operations listed in §8.2.4.

Additionally, `NVECTOR_MPIMANYVECTOR` sets no limit on the number of subvectors that may be attached (aside from the limitations of using `sunindextype` for indexing, and standard per-node memory limitations). However, while this ostensibly supports subvectors with one entry each (i.e., one subvector for each solution entry), we anticipate that this extreme situation will hinder performance due to non-stride-one memory accesses and increased function call overhead. We therefore recommend a relatively coarse partitioning of the problem, although actual performance will likely be problem-dependent.

As a final note, in the coming years we plan to introduce additional algebraic solvers and time integration modules that will leverage the problem partitioning enabled by `NVECTOR_MPIMANYVECTOR`. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side function, DAE or nonlinear solver residual function, preconditioners, or custom `SUNLinearSolver` or `SUNNonlinearSolver` modules.

### 8.18.1 NVECTOR\_MPIMANYVECTOR structure

The NVECTOR\_MPIMANYVECTOR implementation defines the *content* field of `N_Vector` to be a structure containing the MPI communicator (or `MPI_COMM_NULL` if running on a single-node), the number of subvectors comprising the MPIManyVector, the global length of the MPIManyVector (including all subvectors on all MPI ranks), a pointer to the beginning of the array of subvectors, and a boolean flag `own_data` indicating ownership of the subvectors that populate `subvec_array`.

```
struct _N_VectorContent_MPIManyVector {
    MPI_Comm      comm;           /* overall MPI communicator      */
    sunindextype  num_subvectors; /* number of vectors attached    */
    sunindextype  global_length;  /* overall mpimanyvector length  */
    N_Vector*     subvec_array;   /* pointer to N_Vector array     */
    sunbooleantype own_data;      /* flag indicating data ownership */
};
```

The header file to include when using this module is `nvector_mpimanyvector.h`. The installed module library to link against is `libsundials_nvecmpimanyvector.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

#### Note

If SUNDIALS is configured with MPI disabled, then the MPIManyVector library will not be built. Furthermore, any user codes that include `nvector_mpimanyvector.h` *must* be compiled using an MPI-aware compiler (whether the specific user code utilizes MPI or not). We note that the NVECTOR\_MANYVECTOR implementation is designed for ManyVector use cases in an MPI-unaware environment.

### 8.18.2 NVECTOR\_MPIMANYVECTOR functions

The NVECTOR\_MPIMANYVECTOR module implements all vector operations listed in §8.2, except for `N_VGetArrayPointer()`, `N_VSetArrayPointer()`, `N_VScaleAddMultiVectorArray()`, and `N_VLinearCombinationVectorArray()`. As such, this vector cannot be used with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR\_MPIMANYVECTOR module provides functions to access subvectors, whose data may in turn be accessed according to their NVECTOR implementations.

The names of vector operations are obtained from those in §8.2 by appending the suffix `_MPIManyVector` (e.g. `N_VDestroy_MPIManyVector`). The module NVECTOR\_MPIMANYVECTOR provides the following additional user-callable routines:

**`N_Vector N_VNew_MPIManyVector(sunindextype num_subvectors, N_Vector *vec_array, SUNContext sunctx)`**

This function creates a MPIManyVector from a set of existing NVECTOR objects, under the requirement that all MPI-aware subvectors use the same MPI communicator (this is checked internally). If none of the subvectors are MPI-aware, then this may equivalently be used to describe data partitioning within a single node. We note that this routine is designed to support use cases A and C above.

This routine will copy all `N_Vector` pointers from the input `vec_array`, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns NULL (e.g., if two MPI-aware subvectors use different MPI communicators).

Users of the Fortran 2003 interface to this function will first need to use the generic `N_Vector` utility functions `N_VNewVectorArray()`, and `N_VSetVecAtIndexVectorArray()` to create the `N_Vector*` argument. This is further explained in §20.3.5, and the functions are documented in §8.1.1.



*N\_Vector* **N\_VMake\_MPIManyVector**(MPI\_Comm comm, *sunindextype* num\_subvectors, *N\_Vector* \*vec\_array, *SUNContext* sunctx)

This function creates a MPIManyVector from a set of existing NVECTOR objects, and a user-created MPI communicator that “connects” these subvectors. Any MPI-aware subvectors may use different MPI communicators than the input *comm*. We note that this routine is designed to support any combination of the use cases above.

The input *comm* should be this user-created MPI communicator. This routine will internally call `MPI_Comm_dup` to create a copy of the input *comm*, so the user-supplied *comm* argument need not be retained after the call to `N_VMake_MPIManyVector()`.

If all subvectors are MPI-unaware, then the input *comm* argument should be `MPI_COMM_NULL`, although in this case, it would be simpler to call `N_VNew_MPIManyVector()` instead, or to just use the NVECTOR\_MANYVECTOR module.

This routine will copy all *N\_Vector* pointers from the input *vec\_array*, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns `NULL` (e.g., if the input *vec\_array* is `NULL`).

*N\_Vector* **N\_VGetSubvector\_MPIManyVector**(*N\_Vector* v, *sunindextype* vec\_num)

This function returns the *vec\_num* subvector from the NVECTOR array.

*sunindextype* **N\_VGetSubvectorLocalLength\_MPIManyVector**(*N\_Vector* v, *sunindextype* vec\_num)

This function returns the local length of the *vec\_num* subvector from the NVECTOR array.

Usage:

```
local_length = N_VGetSubvectorLocalLength_MPIManyVector(v, 0);
```

*sunrealtype* \***N\_VGetSubvectorArrayPointer\_MPIManyVector**(*N\_Vector* v, *sunindextype* vec\_num)

This function returns the data array pointer for the *vec\_num* subvector from the NVECTOR array.

If the input *vec\_num* is invalid, or if the subvector does not support the `N_VGetArrayPointer` operation, then `NULL` is returned.

*SUNErrCode* **N\_VSetSubvectorArrayPointer\_MPIManyVector**(*sunrealtype* \*v\_data, *N\_Vector* v, *sunindextype* vec\_num)

This function sets the data array pointer for the *vec\_num* subvector from the NVECTOR array.

The function returns a *SUNErrCode*.

*sunindextype* **N\_VGetNumSubvectors\_MPIManyVector**(*N\_Vector* v)

This function returns the overall number of subvectors in the MPIManyVector object.

By default all fused and vector array operations are disabled in the NVECTOR\_MPIMANYVECTOR module, except for `N_VWrmsNormVectorArray()` and `N_VWrmsNormMaskVectorArray()`, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_MPIManyVector()` or `N_VMake_MPIManyVector()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with `N_VNew_MPIManyVector()` and `N_VMake_MPIManyVector()` will have the default settings for the NVECTOR\_MPIMANYVECTOR module. We note that these routines *do not* call the corresponding routines on subvectors, so those should be set up as desired *before* attaching them to the MPIManyVector in `N_VNew_MPIManyVector()` or `N_VMake_MPIManyVector()`.

*SUNErrCode* **N\_VEnableFusedOps\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the MPI-ManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearCombination\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the MPI-ManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleAddMulti\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the MPIManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableDotProdMulti\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the MPI-ManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableLinearSumVectorArray\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the MPI-ManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableScaleVectorArray\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the MPI-ManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableConstVectorArray\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the MPI-ManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormVectorArray\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the MPIManyVector vector. The return value is a *SUNErrCode*.

*SUNErrCode* **N\_VEnableWrmsNormMaskVectorArray\_MPIManyVector**(*N\_Vector* v, *sunbooleantype* tf)

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the MPIManyVector vector. The return value is a *SUNErrCode*.

#### Notes

- *N\_VNew\_MPIManyVector()* and *N\_VMake\_MPIManyVector()* set the field `own_data` = SUNFALSE. The MPI-ManyVector implementation of *N\_VDestroy()* will not attempt to call *N\_VDestroy()* on any subvectors contained in the subvector array for any *N\_Vector* with `own_data` set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the subvectors.
- To maximize efficiency, arithmetic vector operations in the NVECTOR\_MPIMANYVECTOR implementation that have more than one *N\_Vector* argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with *N\_Vector* arguments that were all created with the same subvector representations.

## 8.19 The NVECTOR\_MPIPLUSX Module

The NVECTOR\_MPIPLUSX module is designed to facilitate the MPI+X paradigm, where X is some form of on-node (local) parallelism (e.g. OpenMP, CUDA). This paradigm is becoming increasingly popular with the rise of heterogeneous computing architectures.

The NVECTOR\_MPIPLUSX implementation is designed to work with any NVECTOR that implements the minimum “standard” set of operations in §8.2.1. However, it is not recommended to use the NVECTOR\_PARALLEL, NVECTOR\_PARHYP, NVECTOR\_PETSC, or NVECTOR\_TRILINOS implementations underneath the NVECTOR\_MPIPLUSX module since they already provide MPI capabilities.

### 8.19.1 NVECTOR\_MPIPLUSX structure

The NVECTOR\_MPIPLUSX implementation is a thin wrapper around the NVECTOR\_MPIMANYVECTOR. Accordingly, it adopts the same content structure as defined in §8.18.1.

The header file to include when using this module is `nvector_mpiplusx.h`. The installed module library to link against is `libsundials_nvecmpiplusx.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

#### Note

If SUNDIALS is configured with MPI disabled, then the `mpiplusx` library will not be built. Furthermore, any user codes that include `nvector_mpiplusx.h` *must* be compiled using an MPI-aware compiler.

### 8.19.2 NVECTOR\_MPIPLUSX functions

The NVECTOR\_MPIPLUSX module adopts all vector operations listed in §8.2, from the NVECTOR\_MPIMANYVECTOR (see §8.18) except for `N_VGetArrayPointer()`, and `N_VSetArrayPointer()`; the module provides its own implementation of these functions that call the local vector implementations. Therefore, the NVECTOR\_MPIPLUSX module implements all of the operations listed in the referenced sections except for `N_VScaleAddMultiVectorArray()`, and `N_VLinearCombinationVectorArray()`. Accordingly, its compatibility with the SUNDIALS direct solvers and preconditioners depends on the local vector implementation.

The module NVECTOR\_MPIPLUSX provides the following additional user-callable routines:

`N_Vector` **N\_VMake\_MPIPlusX**(MPI\_Comm comm, `N_Vector` \*local\_vector, `SUNContext` sunctx)

This function creates a MPIPlusX vector from an existing local (i.e. on node) NVECTOR object, and a user-created MPI communicator.

The input *comm* should be this user-created MPI communicator. This routine will internally call `MPI_Comm_dup` to create a copy of the input *comm*, so the user-supplied *comm* argument need not be retained after the call to `N_VMake_MPIPlusX()`.

This routine will copy the NVECTOR pointer to the input *local\_vector*, so the underlying local NVECTOR object should not be destroyed before the `mpiplusx` that contains it.

Upon successful completion, the new MPIPlusX is returned; otherwise this routine returns NULL (e.g., if the input *local\_vector* is NULL).

`N_Vector` **N\_VGetLocalVector\_MPIPlusX**(`N_Vector` v)

This function returns the local vector underneath the MPIPlusX NVECTOR.

`sunindextype` **N\_VGetLocalLength\_MPIPlusX**(`N_Vector` v)

This function returns the local length of the vector underneath the MPIPlusX NVECTOR.

Usage:

```
local_length = N_VGetLocalLength_MPIPlusX(v);
```

*sunrealtype* \***N\_VGetArrayPointer\_MPIPlusX**(*N\_Vector* v)

This function returns the data array pointer for the local vector.

If the local vector does not support the *N\_VGetArrayPointer()* operation, then NULL is returned.

void **N\_VSetArrayPointer\_MPIPlusX**(*sunrealtype* \*v\_data, *N\_Vector* v)

This function sets the data array pointer for the local vector if the local vector implements the *N\_VSetArrayPointer()* operation.

The NVECTOR\_MPIPLUSX module does not implement any fused or vector array operations. Instead users should enable/disable fused operations on the local vector.

#### Notes

- *N\_VMake\_MPIPlusX()* sets the field `own_data = SUNFALSE` and the MPIPlusX implementation of *N\_VDestroy()* will not call *N\_VDestroy()* on the local vector. In this a case, it is the user's responsibility to deallocate the local vector.
- To maximize efficiency, arithmetic vector operations in the NVECTOR\_MPIPLUSX implementation that have more than one *N\_Vector* argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with *N\_Vector* arguments that were all created with the same subvector representations.

## 8.20 NVECTOR Examples

There are NVECTOR examples that may be installed for each implementation. Each implementation makes use of the functions in `test_nvector.c`. These example functions show simple usage of the NVECTOR family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag.

The following is a list of the example functions in `test_nvector.c`:

- `Test_N_VClone`: Creates clone of vector and checks validity of clone.
- `Test_N_VCloneEmpty`: Creates clone of empty vector and checks validity of clone.
- `Test_N_VCloneVectorArray`: Creates clone of vector array and checks validity of cloned array.
- `Test_N_VCloneVectorArray`: Creates clone of empty vector array and checks validity of cloned array.
- `Test_N_VGetArrayPointer`: Get array pointer.
- `Test_N_VSetArrayPointer`: Allocate new vector, set pointer to new vector array, and check values.
- `Test_N_VGetLength`: Compares self-reported length to calculated length.
- `Test_N_VGetCommunicator`: Compares self-reported communicator to the one used in constructor; or for MPI-unaware vectors it ensures that NULL is reported.
- `Test_N_VLinearSum Case 1a`: Test  $y = x + y$
- `Test_N_VLinearSum Case 1b`: Test  $y = -x + y$
- `Test_N_VLinearSum Case 1c`: Test  $y = ax + y$
- `Test_N_VLinearSum Case 2a`: Test  $x = x + y$
- `Test_N_VLinearSum Case 2b`: Test  $x = x - y$
- `Test_N_VLinearSum Case 2c`: Test  $x = x + by$
- `Test_N_VLinearSum Case 3`: Test  $z = x + y$

- Test\_N\_VLinearSum Case 4a: Test  $z = x - y$
  - Test\_N\_VLinearSum Case 4b: Test  $z = -x + y$
  - Test\_N\_VLinearSum Case 5a: Test  $z = x + by$
  - Test\_N\_VLinearSum Case 5b: Test  $z = ax + y$
  - Test\_N\_VLinearSum Case 6a: Test  $z = -x + by$
  - Test\_N\_VLinearSum Case 6b: Test  $z = ax - y$
  - Test\_N\_VLinearSum Case 7: Test  $z = a(x + y)$
  - Test\_N\_VLinearSum Case 8: Test  $z = a(x - y)$
  - Test\_N\_VLinearSum Case 9: Test  $z = ax + by$
  - Test\_N\_VConst: Fill vector with constant and check result.
  - Test\_N\_VProd: Test vector multiply:  $z = x * y$
  - Test\_N\_VDiv: Test vector division:  $z = x / y$
  - Test\_N\_VScale: Case 1: scale:  $x = cx$
  - Test\_N\_VScale: Case 2: copy:  $z = x$
  - Test\_N\_VScale: Case 3: negate:  $z = -x$
  - Test\_N\_VScale: Case 4: combination:  $z = cx$
  - Test\_N\_VAbs: Create absolute value of vector.
  - Test\_N\_VInv: Compute  $z[i] = 1 / x[i]$
- \*\* Test\_N\_VAddConst: add constant vector:  $z = c + x$
- Test\_N\_VDotProd: Calculate dot product of two vectors.
  - Test\_N\_VMaxNorm: Create vector with known values, find and validate the max norm.
  - Test\_N\_VWrmsNorm: Create vector of known values, find and validate the weighted root mean square.
  - Test\_N\_VWrmsNormMask: Create vector of known values, find and validate the weighted root mean square using all elements except one.
  - Test\_N\_VMin: Create vector, find and validate the min.
  - Test\_N\_VWL2Norm: Create vector, find and validate the weighted Euclidean L2 norm.
  - Test\_N\_VL1Norm: Create vector, find and validate the L1 norm.
  - Test\_N\_VCompare: Compare vector with constant returning and validating comparison vector.
  - Test\_N\_VInvTest: Test  $z[i] = 1 / x[i]$
  - Test\_N\_VConstrMask: Test mask of vector  $x$  with vector  $c$ .
  - Test\_N\_VMinQuotient: Fill two vectors with known values. Calculate and validate minimum quotient.
  - Test\_N\_VLinearCombination: Case 1a: Test  $x = a x$
  - Test\_N\_VLinearCombination: Case 1b: Test  $z = a x$
  - Test\_N\_VLinearCombination: Case 2a: Test  $x = a x + b y$
  - Test\_N\_VLinearCombination: Case 2b: Test  $z = a x + b y$
  - Test\_N\_VLinearCombination: Case 3a: Test  $x = x + a y + b z$

- Test\_N\_VLinearCombination: Case 3b: Test  $x = a x + b y + c z$
- Test\_N\_VLinearCombination: Case 3c: Test  $w = a x + b y + c z$
- Test\_N\_VScaleAddMulti: Case 1a:  $y = a x + y$
- Test\_N\_VScaleAddMulti: Case 1b:  $z = a x + y$
- Test\_N\_VScaleAddMulti: Case 2a:  $Y[i] = c[i] x + Y[i]$ ,  $i = 1, 2, 3$
- Test\_N\_VScaleAddMulti: Case 2b:  $Z[i] = c[i] x + Y[i]$ ,  $i = 1, 2, 3$
- Test\_N\_VDotProdMulti: Case 1: Calculate the dot product of two vectors
- Test\_N\_VDotProdMulti: Case 2: Calculate the dot product of one vector with three other vectors in a vector array.
- Test\_N\_VLinearSumVectorArray: Case 1:  $z = a x + b y$
- Test\_N\_VLinearSumVectorArray: Case 2a:  $Z[i] = a X[i] + b Y[i]$
- Test\_N\_VLinearSumVectorArray: Case 2b:  $X[i] = a X[i] + b Y[i]$
- Test\_N\_VLinearSumVectorArray: Case 2c:  $Y[i] = a X[i] + b Y[i]$
- Test\_N\_VScaleVectorArray: Case 1a:  $y = c y$
- Test\_N\_VScaleVectorArray: Case 1b:  $z = c y$
- Test\_N\_VScaleVectorArray: Case 2a:  $Y[i] = c[i] Y[i]$
- Test\_N\_VScaleVectorArray: Case 2b:  $Z[i] = c[i] Y[i]$
- Test\_N\_VConstVectorArray: Case 1a:  $z = c$
- Test\_N\_VConstVectorArray: Case 1b:  $Z[i] = c$
- Test\_N\_VWrmsNormVectorArray: Case 1a: Create a vector of know values, find and validate the weighted root mean square norm.
- Test\_N\_VWrmsNormVectorArray: Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each.
- Test\_N\_VWrmsNormMaskVectorArray: Case 1a: Create a vector of know values, find and validate the weighted root mean square norm using all elements except one.
- Test\_N\_VWrmsNormMaskVectorArray: Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each using all elements except one.
- Test\_N\_VScaleAddMultiVectorArray: Case 1a:  $y = a x + y$
- Test\_N\_VScaleAddMultiVectorArray: Case 1b:  $z = a x + y$
- Test\_N\_VScaleAddMultiVectorArray: Case 2a:  $Y[j][0] = a[j] X[0] + Y[j][0]$
- Test\_N\_VScaleAddMultiVectorArray: Case 2b:  $Z[j][0] = a[j] X[0] + Y[j][0]$
- Test\_N\_VScaleAddMultiVectorArray: Case 3a:  $Y[0][i] = a[0] X[i] + Y[0][i]$
- Test\_N\_VScaleAddMultiVectorArray: Case 3b:  $Z[0][i] = a[0] X[i] + Y[0][i]$
- Test\_N\_VScaleAddMultiVectorArray: Case 4a:  $Y[j][i] = a[j] X[i] + Y[j][i]$
- Test\_N\_VScaleAddMultiVectorArray: Case 4b:  $Z[j][i] = a[j] X[i] + Y[j][i]$
- Test\_N\_VLinearCombinationVectorArray: Case 1a:  $x = a x$
- Test\_N\_VLinearCombinationVectorArray: Case 1b:  $z = a x$
- Test\_N\_VLinearCombinationVectorArray: Case 2a:  $x = a x + b y$

- `Test_N_VLinearCombinationVectorArray`: Case 2b:  $z = a x + b y$
- `Test_N_VLinearCombinationVectorArray`: Case 3a:  $x = a x + b y + c z$
- `Test_N_VLinearCombinationVectorArray`: Case 3b:  $w = a x + b y + c z$
- `Test_N_VLinearCombinationVectorArray`: Case 4a:  $X[0][i] = c[0] X[0][i]$
- `Test_N_VLinearCombinationVectorArray`: Case 4b:  $Z[i] = c[0] X[0][i]$
- `Test_N_VLinearCombinationVectorArray`: Case 5a:  $X[0][i] = c[0] X[0][i] + c[1] X[1][i]$
- `Test_N_VLinearCombinationVectorArray`: Case 5b:  $Z[i] = c[0] X[0][i] + c[1] X[1][i]$
- `Test_N_VLinearCombinationVectorArray`: Case 6a:  $X[0][i] = X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- `Test_N_VLinearCombinationVectorArray`: Case 6b:  $X[0][i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- `Test_N_VLinearCombinationVectorArray`: Case 6c:  $Z[i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- `Test_N_VDotProdLocal`: Calculate MPI task-local portion of the dot product of two vectors.
- `Test_N_VMaxNormLocal`: Create vector with known values, find and validate the MPI task-local portion of the max norm.
- `Test_N_VMinLocal`: Create vector, find and validate the MPI task-local min.
- `Test_N_VL1NormLocal`: Create vector, find and validate the MPI task-local portion of the L1 norm.
- `Test_N_VWSqrSumLocal`: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors.
- `Test_N_VWSqrSumMaskLocal`: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors, using all elements except one.
- `Test_N_VInvTestLocal`: Test the MPI task-local portion of  $z[i] = 1 / x[i]$
- `Test_N_VConstrMaskLocal`: Test the MPI task-local portion of the mask of vector  $x$  with vector  $c$ .
- `Test_N_VMinQuotientLocal`: Fill two vectors with known values. Calculate and validate the MPI task-local minimum quotient.
- `Test_N_VBufSize`: Tests for accuracy in the reported buffer size.
- `Test_N_VBufPack`: Tests for accuracy in the buffer packing routine.
- `Test_N_VBufUnpack`: Tests for accuracy in the buffer unpacking routine.





## Chapter 9

# 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 *SUNLinearSolver* implementations that factor these matrix objects and use them in the solution of linear systems.

### 9.1 Description of the SUNMATRIX Modules

For problems that involve direct methods for solving linear systems, the SUNDIALS packages 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. The generic *SUNMatrix* operations are described below, and descriptions of the SUNMATRIX 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.

A *SUNMatrix* is a pointer to the *\_generic\_SUNMatrix* structure:

```
typedef struct _generic_SUNMatrix *SUNMatrix
```

```
struct _generic_SUNMatrix
```

The structure defining the SUNDIALS matrix class.

void \***content**

Pointer to matrix-specific member data

```
struct _generic_SUNMatrix_Ops *ops
```

A virtual table of matrix operations provided by a specific implementation

*SUNContext* **sunctx**

The SUNDIALS simulation context

The virtual table structure is defined as

struct **\_generic\_SUNMatrix\_Ops**

The structure defining *SUNMatrix* operations.

*SUNMatrix\_ID* (\***getid**)(*SUNMatrix*)

The function implementing *SUNMatGetID()*

*SUNMatrix* (\***clone**)(*SUNMatrix*)

The function implementing *SUNMatClone()*

void (\***destroy**)(*SUNMatrix*)

The function implementing *SUNMatDestroy()*

*SUNErrCode* (\***zero**)(*SUNMatrix*)

The function implementing *SUNMatZero()*

*SUNErrCode* (\***copy**)(*SUNMatrix*, *SUNMatrix*)

The function implementing *SUNMatCopy()*

*SUNErrCode* (\***scaleadd**)(*sunrealtype*, *SUNMatrix*, *SUNMatrix*)

The function implementing *SUNMatScaleAdd()*

*SUNErrCode* (\***scaleaddi**)(*sunrealtype*, *SUNMatrix*)

The function implementing *SUNMatScaleAddI()*

*SUNErrCode* (\***matvecsetup**)(*SUNMatrix*)

The function implementing *SUNMatMatvecSetup()*

*SUNErrCode* (\***matvec**)(*SUNMatrix*, *N\_Vector*, *N\_Vector*)

The function implementing *SUNMatMatvec()*

*SUNErrCode* (\***mathhermitiantransposevec**)(*SUNMatrix*, *N\_Vector*, *N\_Vector*)

The function implementing *SUNMatHermitianTransposeVec()*

Added in version 7.3.0.

*SUNErrCode* (\***space**)(*SUNMatrix*, long int\*, long int\*)

The function implementing *SUNMatSpace()*

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:

```
SUNErrCode SUNMatZero(SUNMatrix A)
{
    return(A->ops->zero(A));
}
```

§9.2 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 package and/or linear solver to determine which SUNMATRIX operations they require.

Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMATRIX module (each with different *SUNMatrix* internal data representations) in the same code.

- Define and implement user-callable constructor and destructor routines to create and free a `SUNMatrix` with the new *content* field and with *ops* pointing to the new matrix operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNMatrix` (e.g., a routine to print the *content* for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the content field of the newly defined `SUNMatrix`.

To aid in the creation of custom `SUNMATRIX` modules the generic `SUNMATRIX` module provides three utility functions `SUNMatNewEmpty()`, `SUNMatCopyOps()`, and `SUNMatFreeEmpty()`. When used in custom `SUNMATRIX` constructors and clone routines these functions will ease the introduction of any new optional matrix operations to the `SUNMATRIX` API by ensuring only required operations need to be set and all operations are copied when cloning a matrix.

*SUNMatrix* **SUNMatNewEmpty**(*SUNContext* suncctx)

This function allocates a new generic `SUNMatrix` object and initializes its content pointer and the function pointers in the operations structure to NULL.

**Return value:**

If successful, this function returns a `SUNMatrix` object. If an error occurs when allocating the object, then this routine will return NULL.

*SUNErrCode* **SUNMatCopyOps**(*SUNMatrix* A, *SUNMatrix* B)

This function copies the function pointers in the *ops* structure of A into the *ops* structure of B.

**Arguments:**

- A – the matrix to copy operations from.
- B – the matrix to copy operations to.

**Return value:**

- A *SUNErrCode*

void **SUNMatFreeEmpty**(*SUNMatrix* A)

This routine frees the generic `SUNMatrix` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the *ops* pointer is NULL, and, if it is not, it will free it as well.

**Arguments:**

- A – the `SUNMatrix` object to free

type **SUNMatrix\_ID**

Each `SUNMATRIX` implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 9.1. It is recommended that a user-supplied `SUNMATRIX` implementation use the `SUNMATRIX_CUSTOM` identifier.

Table 9.1: Identifiers associated with matrix kernels supplied with SUN-DIALS

Matrix ID	Matrix type
SUNMATRIX_BAND	Band $M \times M$ matrix
SUNMATRIX_CUSPARSE	CUDA sparse CSR matrix
SUNMATRIX_CUSTOM	User-provided custom matrix
SUNMATRIX_DENSE	Dense $M \times N$ matrix
SUNMATRIX_GINKGO	SUNMatrix wrapper for Ginkgo matrices
SUNMATRIX_MAGMADENSE	Dense $M \times N$ matrix
SUNMATRIX_ONEMKLDENSE	oneMKL dense $M \times N$ matrix
SUNMATRIX_SLUNRLOC	SUNMatrix wrapper for SuperLU_DIST SuperMatrix
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $M \times N$ matrix

## 9.2 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 Table 9.1

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 values, 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);
```

*SUNErrCode* **SUNMatSpace**(*SUNMatrix* A, long int \*lrw, long int \*liw)

Returns the storage requirements for the matrix *A*. *lrw* contains the number of sunrealtype 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:

```
retval = SUNMatSpace(A, &lrw, &liw);
```

Deprecated since version 7.3.0: Work space functions will be removed in version 8.0.0.

**SUNErrCode SUNMatZero(SUNMatrix A)**

Zeros all entries of the SUNMatrix  $A$ . The return value denotes the success/failure of the operation:

$$A_{i,j} = 0, \quad i = 1, \dots, m, \quad j = 1, \dots, n.$$

Usage:

```
retval = SUNMatZero(A);
```

**SUNErrCode SUNMatCopy(SUNMatrix A, SUNMatrix B)**

Performs the operation  $B$  gets  $A$  for all entries of the matrices  $A$  and  $B$ . The return value denotes the success/failure of the operation:

$$B_{i,j} = A_{i,j}, \quad i = 1, \dots, m, \quad j = 1, \dots, n.$$

Usage:

```
retval = SUNMatCopy(A,B);
```

**SUNErrCode SUNMatScaleAdd(sunrealtype c, SUNMatrix A, SUNMatrix B)**

Performs the operation  $A$  gets  $cA + B$ . The return value denotes the success/failure of the operation:

$$A_{i,j} = cA_{i,j} + B_{i,j}, \quad i = 1, \dots, m, \quad j = 1, \dots, n.$$

Usage:

```
retval = SUNMatScaleAdd(c, A, B);
```

**SUNErrCode SUNMatScaleAddI(sunrealtype c, SUNMatrix A)**

Performs the operation  $A$  gets  $cA + I$ . The return value denotes the success/failure of the operation:

$$A_{i,j} = cA_{i,j} + \delta_{i,j}, \quad i, j = 1, \dots, n.$$

Usage:

```
retval = SUNMatScaleAddI(c, A);
```

**SUNErrCode SUNMatMatvecSetup(SUNMatrix A)**

Performs any setup necessary to perform a matrix-vector product. The return value denotes the success/failure of the operation. It is useful for SUNMatrix implementations which need to prepare the matrix itself, or communication structures before performing the matrix-vector product.

Usage:

```
retval = SUNMatMatvecSetup(A);
```

**SUNErrCode SUNMatMatvec(SUNMatrix A, N\_Vector x, N\_Vector y)**

Performs the matrix-vector product  $y \leftarrow Ax$ . 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 denotes the success/failure of the operation:

$$y_i = \sum_{j=1}^n A_{i,j}x_j, \quad i = 1, \dots, m.$$

Usage:

```
retval = SUNMatMatvec(A, x, y);
```

**SUNErrCode SUNMatHermitianTransposeVec**(*SUNMatrix* A, *N\_Vector* x, *N\_Vector* y)

Performs the matrix-vector product  $y \leftarrow A^*x$  where  $*$  is the Hermitian (conjugate) transpose. 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 denotes the success/failure of the operation:

$$y_i = \sum_{j=1}^n \bar{A}_{j,i} x_j, \quad i = 1, \dots, m.$$

where  $\bar{c}$  denotes the complex conjugate of  $c$ .

Usage:

```
retval = SUNMatHermitianTransposeVec(A, x, y);
```

### 9.3 The SUNMATRIX\_DENSE Module

The dense implementation of the *SUNMatrix* module, *SUNMATRIX\_DENSE*, defines the *content* field of *SUNMatrix* to be the following structure:

```
struct _SUNMatrixContent_Dense {
    sunindextype M;
    sunindextype N;
    sunrealtype *data;
    sunindextype ldata;
    sunrealtype **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 *sunrealtype* variables. The elements of the dense matrix are stored columnwise, i.e. the  $(i, j)$  element of a dense *SUNMatrix* object (with  $0 \leq i < M$  and  $0 \leq j < N$ ) may be accessed via `data[j*M+i]`.
- **ldata** - length of the data array ( $= M N$ ).
- **cols** - array of pointers. `cols[j]` points to the first element of the  $j$ -th column of the matrix in the array `data`. The  $(i, j)$  element of a dense *SUNMatrix* (with  $0 \leq i < M$  and  $0 \leq j < N$ ) may be accessed via `cols[j][i]`.

The header file to be included when using this module is `sunmatrix/sunmatrix_dense.h`.

The following macros are provided to access the content of a *SUNMATRIX\_DENSE* matrix. The prefix **SM\_** in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix **\_D** denotes that these are specific to the *dense* version.

**SM\_CONTENT\_D(A)**

This macro gives access to the contents of the dense *SUNMatrix* *A*.

The assignment `A_cont = SM_CONTENT_D(A)` sets `A_cont` to be a pointer to the dense *SUNMatrix* content structure.

Implementation:

```
#define SM_CONTENT_D(A)    ( (SUNMatrixContent_Dense)(A->content) )
```

**SM\_ROWS\_D(A)**

Access the number of rows in the dense SUNMatrix A.

This may be used either to retrieve or to set the value. For example, the assignment `A_rows = SM_ROWS_D(A)` sets `A_rows` to be the number of rows in the matrix A. Similarly, the assignment `SM_ROWS_D(A) = A_rows` sets the number of columns in A to equal `A_rows`.

Implementation:

```
#define SM_ROWS_D(A)      ( SM_CONTENT_D(A)->M )
```

**SM\_COLUMNS\_D(A)**

Access the number of columns in the dense SUNMatrix A.

This may be used either to retrieve or to set the value. For example, the assignment `A_columns = SM_COLUMNS_D(A)` sets `A_columns` to be the number of columns in the matrix A. Similarly, the assignment `SM_COLUMNS_D(A) = A_columns` sets the number of columns in A to equal `A_columns`.

Implementation:

```
#define SM_COLUMNS_D(A)   ( SM_CONTENT_D(A)->N )
```

**SM\_LDATA\_D(A)**

Access the total data length in the dense SUNMatrix A.

This may be used either to retrieve or to set the value. For example, the assignment `A_ldata = SM_LDATA_D(A)` sets `A_ldata` to be the length of the data array in the matrix A. Similarly, the assignment `SM_LDATA_D(A) = A_ldata` sets the parameter for the length of the data array in A to equal `A_ldata`.

Implementation:

```
#define SM_LDATA_D(A)     ( SM_CONTENT_D(A)->ldata )
```

**SM\_DATA\_D(A)**

This macro gives access to the data pointer for the matrix entries.

The assignment `A_data = SM_DATA_D(A)` sets `A_data` to be a pointer to the first component of the data array for the dense SUNMatrix A. The assignment `SM_DATA_D(A) = A_data` sets the data array of A to be `A_data` by storing the pointer `A_data`.

Implementation:

```
#define SM_DATA_D(A)      ( SM_CONTENT_D(A)->data )
```

**SM\_COLS\_D(A)**

This macro gives access to the cols pointer for the matrix entries.

The assignment `A_cols = SM_COLS_D(A)` sets `A_cols` to be a pointer to the array of column pointers for the dense SUNMatrix A. The assignment `SM_COLS_D(A) = A_cols` sets the column pointer array of A to be `A_cols` by storing the pointer `A_cols`.

Implementation:

```
#define SM_COLS_D(A)      ( SM_CONTENT_D(A)->cols )
```

**SM\_COLUMN\_D(A)**

This macros gives access to the individual columns of the data array of a dense SUNMatrix.

The assignment `col_j = SM_COLUMN_D(A, j)` sets `col_j` to be a pointer to the first entry of the  $j$ -th column of the  $M \times N$  dense matrix **A** (with  $0 \leq j < N$ ). The type of the expression `SM_COLUMN_D(A, j)` is `sunrealtype *`. The pointer returned by the call `SM_COLUMN_D(A, j)` can be treated as an array which is indexed from 0 to  $M-1$ .

Implementation:

```
#define SM_COLUMN_D(A,j)    ( (SM_CONTENT_D(A)->cols)[j] )
```

**SM\_ELEMENT\_D(A)**

This macro gives access to the individual entries of the data array of a dense SUNMatrix.

The assignments `SM_ELEMENT_D(A, i, j) = a_ij` and `a_ij = SM_ELEMENT_D(A, i, j)` reference the  $A_{i,j}$  element of the  $M \times N$  dense matrix **A** (with  $0 \leq i < M$  and  $0 \leq j < N$ ).

Implementation:

```
#define SM_ELEMENT_D(A,i,j) ( (SM_CONTENT_D(A)->cols)[j][i] )
```

The `SUNMATRIX_DENSE` module defines dense implementations of all matrix operations listed in §9.2. 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, *SUNContext* sunctx)

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.

*sunrealtype* \***SUNDenseMatrix\_Data**(*SUNMatrix* A)

This function returns a pointer to the data array for the dense SUNMatrix.

*sunrealtype* \*\***SUNDenseMatrix\_Cols**(*SUNMatrix* A)

This function returns a pointer to the cols array for the dense SUNMatrix.

*sunrealtype* \***SUNDenseMatrix\_Column**(*SUNMatrix* A, *sunindextype* j)

This function returns a pointer to the first entry of the  $j$ th 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 = SUNDenseMatrix_Data(A)`, or equivalently `A_data = SM_DATA_D(A)`, and then access `A_data[i]` within the loop.



- First obtain the array of column pointers via `A_cols = SUNDenseMatrix_Cols(A)`, or equivalently `A_cols = SM_COLS_D(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.

## 9.4 The SUNMATRIX\_MAGMADENSE Module

The `SUNMATRIX_MAGMADENSE` module interfaces to the [MAGMA](#) linear algebra library and can target NVIDIA's CUDA programming model or AMD's HIP programming model [117]. All data stored by this matrix implementation resides on the GPU at all times. The implementation currently supports a standard LAPACK column-major storage format as well as a low-storage format for block-diagonal matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix}$$

This matrix implementation is best paired with the *SUNLinearSolver\_MagmaDense* `SUNLinearSolver`.

The header file to include when using this module is `sunmatrix/sunmatrix_magmadense.h`. The installed library to link to is `libsundials_sunmatrixmagmadense.lib` where `lib` is typically `.so` for shared libraries and `.a` for static libraries.

### Warning

The `SUNMATRIX_MAGMADENSE` module is experimental and subject to change.

### 9.4.1 SUNMATRIX\_MAGMADENSE Functions

The `SUNMATRIX_MAGMADENSE` module defines GPU-enabled implementations of all matrix operations listed in §9.2.

- `SUNMatGetID_MagmaDense` – returns `SUNMATRIX_MAGMADENSE`
- `SUNMatClone_MagmaDense`
- `SUNMatDestroy_MagmaDense`
- `SUNMatZero_MagmaDense`
- `SUNMatCopy_MagmaDense`
- `SUNMatScaleAdd_MagmaDense`
- `SUNMatScaleAddI_MagmaDense`
- `SUNMatMatvecSetup_MagmaDense`
- `SUNMatMatvec_MagmaDense`

- `SUNMatSpace_MagmaDense`

In addition, the `SUNMATRIX_MAGMADENSE` module defines the following implementation specific functions:

*SUNMatrix* **`SUNMatrix_MagmaDense`**(*sunindextype* M, *sunindextype* N, *SUNMemoryType* memtype, *SUNMemoryHelper* memhelper, void \*queue, *SUNContext* suctx)

This constructor function creates and allocates memory for an  $M \times N$  `SUNMATRIX_MAGMADENSE` `SUNMatrix`.

**Arguments:**

- $M$  – the number of matrix rows.
- $N$  – the number of matrix columns.
- *memtype* – the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
- *memhelper* – the memory helper used for allocating data.
- *queue* – a `cudaStream_t` when using CUDA or a `hipStream_t` when using HIP.
- *suctx* – the *SUNContext* object (see §4.2)

**Return value:**

If successful, a `SUNMatrix` object otherwise `NULL`.

*SUNMatrix* **`SUNMatrix_MagmaDenseBlock`**(*sunindextype* nblocks, *sunindextype* M\_block, *sunindextype* N\_block, *SUNMemoryType* memtype, *SUNMemoryHelper* memhelper, void \*queue, *SUNContext* suctx)

This constructor function creates and allocates memory for a block diagonal `SUNMATRIX_MAGMADENSE` `SUNMatrix` with *nblocks* of size  $M \times N$ .

**Arguments:**

- *nblocks* – the number of matrix rows.
- *M\_block* – the number of matrix rows in each block.
- *N\_block* – the number of matrix columns in each block.
- *memtype* – the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
- *memhelper* – the memory helper used for allocating data.
- *queue* – a `cudaStream_t` when using CUDA or a `hipStream_t` when using HIP.
- *suctx* – the *SUNContext* object (see §4.2)

**Return value:**

If successful, a `SUNMatrix` object otherwise `NULL`.

*sunindextype* **`SUNMatrix_MagmaDense_Rows`**(*SUNMatrix* A)

This function returns the number of rows in the `SUNMatrix` object. For block diagonal matrices, the number of rows is computed as  $M_{\text{block}} \times \text{nblocks}$ .

**Arguments:**

- *A* – a `SUNMatrix` object.

**Return value:**

If successful, the number of rows in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

*sunindex*type **SUNMatrix\_MagmaDense\_Columns**(*SUNMatrix* A)

This function returns the number of columns in the *SUNMatrix* object. For block diagonal matrices, the number of columns is computed as  $N_{\text{block}} \times \text{nblocks}$ .

**Arguments:**

- *A* – a *SUNMatrix* object.

**Return value:**

If successful, the number of columns in the *SUNMatrix* object otherwise `SUNMATRIX_ILL_INPUT`.

*sunindex*type **SUNMatrix\_MagmaDense\_BlockRows**(*SUNMatrix* A)

This function returns the number of rows in a block of the *SUNMatrix* object.

**Arguments:**

- *A* – a *SUNMatrix* object.

**Return value:**

If successful, the number of rows in a block of the *SUNMatrix* object otherwise `SUNMATRIX_ILL_INPUT`.

*sunindex*type **SUNMatrix\_MagmaDense\_BlockColumns**(*SUNMatrix* A)

This function returns the number of columns in a block of the *SUNMatrix* object.

**Arguments:**

- *A* – a *SUNMatrix* object.

**Return value:**

If successful, the number of columns in a block of the *SUNMatrix* object otherwise `SUNMATRIX_ILL_INPUT`.

*sunindex*type **SUNMatrix\_MagmaDense\_LData**(*SUNMatrix* A)

This function returns the length of the *SUNMatrix* data array.

**Arguments:**

- *A* – a *SUNMatrix* object.

**Return value:**

If successful, the length of the *SUNMatrix* data array otherwise `SUNMATRIX_ILL_INPUT`.

*sunindex*type **SUNMatrix\_MagmaDense\_NumBlocks**(*SUNMatrix* A)

This function returns the number of blocks in the *SUNMatrix* object.

**Arguments:**

- *A* – a *SUNMatrix* object.

**Return value:**

If successful, the number of blocks in the *SUNMatrix* object otherwise `SUNMATRIX_ILL_INPUT`.

*sunreal*type **\*SUNMatrix\_MagmaDense\_Data**(*SUNMatrix* A)

This function returns the *SUNMatrix* data array.

**Arguments:**

- *A* – a *SUNMatrix* object.

**Return value:**

If successful, the *SUNMatrix* data array otherwise `NULL`.

*sunrealtype* \*\***SUNMatrix\_MagmaDense\_BlockData**(*SUNMatrix* A)

This function returns an array of pointers that point to the start of the data array for each block in the *SUNMatrix*.

**Arguments:**

- *A* – a *SUNMatrix* object.

**Return value:**

If successful, an array of data pointers to each of the *SUNMatrix* blocks otherwise NULL.

*sunrealtype* \***SUNMatrix\_MagmaDense\_Block**(*SUNMatrix* A, *sunindextype* k)

This function returns a pointer to the data array for block *k* in the *SUNMatrix*.

**Arguments:**

- *A* – a *SUNMatrix* object.
- *k* – the block index.

**Return value:**

If successful, a pointer to the data array for the *SUNMatrix* block otherwise NULL.

**Note**

No bounds-checking is performed by this function, *j* should be strictly less than *nblocks*.

*sunrealtype* \***SUNMatrix\_MagmaDense\_Column**(*SUNMatrix* A, *sunindextype* j)

This function returns a pointer to the data array for column *j* in the *SUNMatrix*.

**Arguments:**

- *A* – a *SUNMatrix* object.
- *j* – the column index.

**Return value:**

If successful, a pointer to the data array for the *SUNMatrix* column otherwise NULL.

**Note**

No bounds-checking is performed by this function, *j* should be strictly less than *nblocks* \*  $N_{\text{block}}$ .

*sunrealtype* \***SUNMatrix\_MagmaDense\_BlockColumn**(*SUNMatrix* A, *sunindextype* k, *sunindextype* j)

This function returns a pointer to the data array for column *j* of block *k* in the *SUNMatrix*.

**Arguments:**

- *A* – a *SUNMatrix* object.
- *k* – the block index.
- *j* – the column index.

**Return value:**

If successful, a pointer to the data array for the *SUNMatrix* column otherwise NULL.

**Note**

No bounds-checking is performed by this function,  $k$  should be strictly less than  $nblocks$  and  $j$  should be strictly less than  $N_{block}$ .

*SUNErrCode* **SUNMatrix\_MagmaDense\_CopyToDevice**(*SUNMatrix* A, *sunrealtype* \*h\_data)

This function copies the matrix data to the GPU device from the provided host array.

**Arguments:**

- $A$  – a *SUNMatrix* object
- $h\_data$  – a host array pointer to copy data from.

**Return value:**

- *SUN\_SUCCESS* – if the copy is successful.
- *SUN\_ERR\_ARG\_INCOMPATIBLE* – if the *SUNMatrix* is not a *SUNMATRIX\_MAGMADENSE* matrix.
- *SUN\_ERR\_MEM\_FAIL* – if the copy fails.

*SUNErrCode* **SUNMatrix\_MagmaDense\_CopyFromDevice**(*SUNMatrix* A, *sunrealtype* \*h\_data)

This function copies the matrix data from the GPU device to the provided host array.

**Arguments:**

- $A$  – a *SUNMatrix* object
- $h\_data$  – a host array pointer to copy data to.

**Return value:**

- *SUN\_SUCCESS* – if the copy is successful.
- *SUN\_ERR\_ARG\_INCOMPATIBLE* – if the *SUNMatrix* is not a *SUNMATRIX\_MAGMADENSE* matrix.
- *SUN\_ERR\_MEM\_FAIL* – if the copy fails.

## 9.4.2 SUNMATRIX\_MAGMADENSE Usage Notes

**Warning**

When using the *SUNMATRIX\_MAGMADENSE* module with a *SUNDIALS* package (e.g. *CVODE*), the stream given to matrix should be the same stream used for the *NVECTOR* object that is provided to the package, and the *NVECTOR* object given to the *SUNMatvec* operation. If different streams are utilized, synchronization issues may occur.

## 9.5 The SUNMATRIX\_ONEMKLDENSE Module

The *SUNMATRIX\_ONEMKLDENSE* module is intended for interfacing with direct linear solvers from the [Intel oneAPI Math Kernel Library \(oneMKL\)](#) using the SYCL (DPC++) programming model. The implementation currently

supports a standard LAPACK column-major storage format as well as a low-storage format for block-diagonal matrices,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix}$$

This matrix implementation is best paired with the *SUNLinearSolver\_OneMklDense* linear solver.

The header file to include when using this class is `sunmatrix/sunmatrix_onemkldense.h`. The installed library to link to is `libsundials_sunmatrixonemkldense.lib` where `lib` is typically `.so` for shared libraries and `.a` for static libraries.

### Warning

The `SUNMATRIX_ONEMKLDENSE` class is experimental and subject to change.

## 9.5.1 SUNMATRIX\_ONEMKLDENSE Functions

The `SUNMATRIX_ONEMKLDENSE` class defines implementations of the following matrix operations listed in §9.2.

- `SUNMatGetID_OneMklDense` – returns `SUNMATRIX_ONEMKLDENSE`
- `SUNMatClone_OneMklDense`
- `SUNMatDestroy_OneMklDense`
- `SUNMatZero_OneMklDense`
- `SUNMatCopy_OneMklDense`
- `SUNMatScaleAdd_OneMklDense`
- `SUNMatScaleAddI_OneMklDense`
- `SUNMatMatvec_OneMklDense`
- `SUNMatSpace_OneMklDense`

In addition, the `SUNMATRIX_ONEMKLDENSE` class defines the following implementation specific functions.

### 9.5.1.1 Constructors

`SUNMatrix` **`SUNMatrix_OneMklDense`**(`sunindextype` `M`, `sunindextype` `N`, `SUNMemoryType` `memtype`, `SUNMemoryHelper` `memhelper`, `sycl::queue` `*queue`, `SUNContext` `sunctx`)

This constructor function creates and allocates memory for an  $M \times N$  `SUNMATRIX_ONEMKLDENSE` `SUNMatrix`.

#### Arguments:

- $M$  – the number of matrix rows.
- $N$  – the number of matrix columns.
- *memtype* – the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
- *memhelper* – the memory helper used for allocating data.

- *queue* – the SYCL queue to which operations will be submitted.
- *sunctx* – the [SUNContext](#) object (see §4.2)

**Return value:**

If successful, a `SUNMatrix` object otherwise `NULL`.

`SUNMatrix` **SUNMatrix\_OneMklDenseBlock**(`sunindextype` nblocks, `sunindextype` M\_block, `sunindextype` N\_block, `SUNMemoryType` memtype, `SUNMemoryHelper` memhelper, `sycl::queue` \*queue, `SUNContext` sunctx)

This constructor function creates and allocates memory for a block diagonal `SUNMATRIX_ONEMKLDENSE` `SUNMatrix` with *nblocks* of size  $M_{block} \times N_{block}$ .

**Arguments:**

- *nblocks* – the number of matrix rows.
- *M\_block* – the number of matrix rows in each block.
- *N\_block* – the number of matrix columns in each block.
- *memtype* – the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
- *memhelper* – the memory helper used for allocating data.
- *queue* – the SYCL queue to which operations will be submitted.
- *sunctx* – the [SUNContext](#) object (see §4.2)

**Return value:**

If successful, a `SUNMatrix` object otherwise `NULL`.

**9.5.1.2 Access Matrix Dimensions**

`sunindextype` **SUNMatrix\_OneMklDense\_Rows**(`SUNMatrix` A)

This function returns the number of rows in the `SUNMatrix` object. For block diagonal matrices, the number of rows is computed as  $M_{block} \times nblocks$ .

**Arguments:**

- *A* – a `SUNMatrix` object.

**Return value:**

If successful, the number of rows in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

`sunindextype` **SUNMatrix\_OneMklDense\_Columns**(`SUNMatrix` A)

This function returns the number of columns in the `SUNMatrix` object. For block diagonal matrices, the number of columns is computed as  $N_{block} \times nblocks$ .

**Arguments:**

- *A* – a `SUNMatrix` object.

**Return value:**

If successful, the number of columns in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

### 9.5.1.3 Access Matrix Block Dimensions

*sunindextype* **SUNMatrix\_OneMklDense\_NumBlocks**(*SUNMatrix* A)

This function returns the number of blocks in the *SUNMatrix* object.

**Arguments:**

- A – a *SUNMatrix* object.

**Return value:**

If successful, the number of blocks in the *SUNMatrix* object otherwise *SUNMATRIX\_ILL\_INPUT*.

*sunindextype* **SUNMatrix\_OneMklDense\_BlockRows**(*SUNMatrix* A)

This function returns the number of rows in a block of the *SUNMatrix* object.

**Arguments:**

- A – a *SUNMatrix* object.

**Return value:**

If successful, the number of rows in a block of the *SUNMatrix* object otherwise *SUNMATRIX\_ILL\_INPUT*.

*sunindextype* **SUNMatrix\_OneMklDense\_BlockColumns**(*SUNMatrix* A)

This function returns the number of columns in a block of the *SUNMatrix* object.

**Arguments:**

- A – a *SUNMatrix* object.

**Return value:**

If successful, the number of columns in a block of the *SUNMatrix* object otherwise *SUNMATRIX\_ILL\_INPUT*.

### 9.5.1.4 Access Matrix Data

*sunindextype* **SUNMatrix\_OneMklDense\_LData**(*SUNMatrix* A)

This function returns the length of the *SUNMatrix* data array.

**Arguments:**

- A – a *SUNMatrix* object.

**Return value:**

If successful, the length of the *SUNMatrix* data array otherwise *SUNMATRIX\_ILL\_INPUT*.

*sunrealtype* \***SUNMatrix\_OneMklDense\_Data**(*SUNMatrix* A)

This function returns the *SUNMatrix* data array.

**Arguments:**

- A – a *SUNMatrix* object.

**Return value:**

If successful, the *SUNMatrix* data array otherwise *NULL*.

*sunrealtype* \***SUNMatrix\_OneMklDense\_Column**(*SUNMatrix* A, *sunindextype* j)

This function returns a pointer to the data array for column *j* in the *SUNMatrix*.

**Arguments:**

- A – a *SUNMatrix* object.
- *j* – the column index.



**Return value:**

If successful, a pointer to the data array for the `SUNMatrix` column otherwise NULL.

**Note**

No bounds-checking is performed by this function,  $j$  should be strictly less than  $nblocks * N_{\text{block}}$ .

**9.5.1.5 Access Matrix Block Data**

*sunindextype* **SUNMatrix\_OneMklDense\_BlockLData**(*SUNMatrix* A)

This function returns the length of the `SUNMatrix` data array for each block of the `SUNMatrix` object.

**Arguments:**

- A – a `SUNMatrix` object.

**Return value:**

If successful, the length of the `SUNMatrix` data array for each block otherwise `SUNMATRIX_ILL_INPUT`.

*sunrealtype* \*\***SUNMatrix\_OneMklDense\_BlockData**(*SUNMatrix* A)

This function returns an array of pointers that point to the start of the data array for each block in the `SUNMatrix`.

**Arguments:**

- A – a `SUNMatrix` object.

**Return value:**

If successful, an array of data pointers to each of the `SUNMatrix` blocks otherwise NULL.

*sunrealtype* \***SUNMatrix\_OneMklDense\_Block**(*SUNMatrix* A, *sunindextype* k)

This function returns a pointer to the data array for block  $k$  in the `SUNMatrix`.

**Arguments:**

- A – a `SUNMatrix` object.
- $k$  – the block index.

**Return value:**

If successful, a pointer to the data array for the `SUNMatrix` block otherwise NULL.

**Note**

No bounds-checking is performed by this function,  $j$  should be strictly less than  $nblocks$ .

*sunrealtype* \***SUNMatrix\_OneMklDense\_BlockColumn**(*SUNMatrix* A, *sunindextype* k, *sunindextype* j)

This function returns a pointer to the data array for column  $j$  of block  $k$  in the `SUNMatrix`.

**Arguments:**

- A – a `SUNMatrix` object.
- $k$  – the block index.
- $j$  – the column index.

**Return value:**

If successful, a pointer to the data array for the `SUNMatrix` column otherwise NULL.

**Note**

No bounds-checking is performed by this function,  $k$  should be strictly less than  $nblocks$  and  $j$  should be strictly less than  $N_{block}$ .

**9.5.1.6 Copy Data**

*SUNErrCode* **SUNMatrix\_OneMklDense\_CopyToDevice**(*SUNMatrix* A, *sunrealtype* \*h\_data)

This function copies the matrix data to the GPU device from the provided host array.

**Arguments:**

- $A$  – a *SUNMatrix* object
- $h\_data$  – a host array pointer to copy data from.

**Return value:**

- *SUN\_SUCCESS* – if the copy is successful.
- *SUN\_ERR\_ARG\_INCOMPATIBLE* – if the *SUNMatrix* is not a *SUNMATRIX\_ONEMKLDENSE* matrix.
- *SUN\_ERR\_MEM\_FAIL* – if the copy fails.

*SUNErrCode* **SUNMatrix\_OneMklDense\_CopyFromDevice**(*SUNMatrix* A, *sunrealtype* \*h\_data)

This function copies the matrix data from the GPU device to the provided host array.

**Arguments:**

- $A$  – a *SUNMatrix* object
- $h\_data$  – a host array pointer to copy data to.

**Return value:**

- *SUN\_SUCCESS* – if the copy is successful.
- *SUN\_ERR\_ARG\_INCOMPATIBLE* – if the *SUNMatrix* is not a *SUNMATRIX\_ONEMKLDENSE* matrix.
- *SUN\_ERR\_MEM\_FAIL* – if the copy fails.

**9.5.2 SUNMATRIX\_ONEMKLDENSE Usage Notes****Warning**

The *SUNMATRIX\_ONEMKLDENSE* class only supports 64-bit indexing, thus *SUNDIALS* must be built for 64-bit indexing to use this class.

When using the *SUNMATRIX\_ONEMKLDENSE* class with a *SUNDIALS* package (e.g. *CVODE*), the queue given to matrix should be the same stream used for the *NVECTOR* object that is provided to the package, and the *NVECTOR* object given to the *SUNMatMatvec()* operation. If different streams are utilized, synchronization issues may occur.

**9.6 The SUNMATRIX\_BAND Module**

The banded implementation of the *SUNMatrix* module, *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;
    sunrealtype *data;
    sunindextype ldata;
    sunrealtype **cols;
};

```

A diagram of the underlying data representation in a banded matrix is shown in Fig. 9.1. 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 [SUNLINSOL\\_BAND](#) and [SUNLINSOL\\_LAPACKBAND](#) modules write the LU factors into the existing storage for the band matrix. The upper triangular factor *U*, however, may have an upper bandwidth as big as  $\min(N-1, \text{mu}+\text{ml})$  because of partial pivoting. The *smu* field holds the upper half-bandwidth allocated for the band matrix.
- *ldim* - leading dimension ( $\text{ldim} \geq \text{smu} + \text{ml} + 1$ )
- *data* - pointer to a contiguous block of `sunrealtype` 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 ( $= \text{ldim } N$ )
- *cols* - array of pointers. *cols*[*j*] is a pointer to the uppermost element within the band in the *j*-th column. This pointer may be treated as an array indexed from *smu*-*mu* (to access the uppermost element within the band in the *j*-th column) to *smu*+*ml* (to access the lowest element within the band in the *j*-th column). Indices from 0 to *smu*-*mu*-1 give access to extra storage elements required by the LU decomposition function. Finally, *cols*[*j*][*i*-*j*+*smu*] is the (*i*, *j*)-th element with  $j - \text{mu} \leq i \leq j + \text{ml}$ .

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) )
```

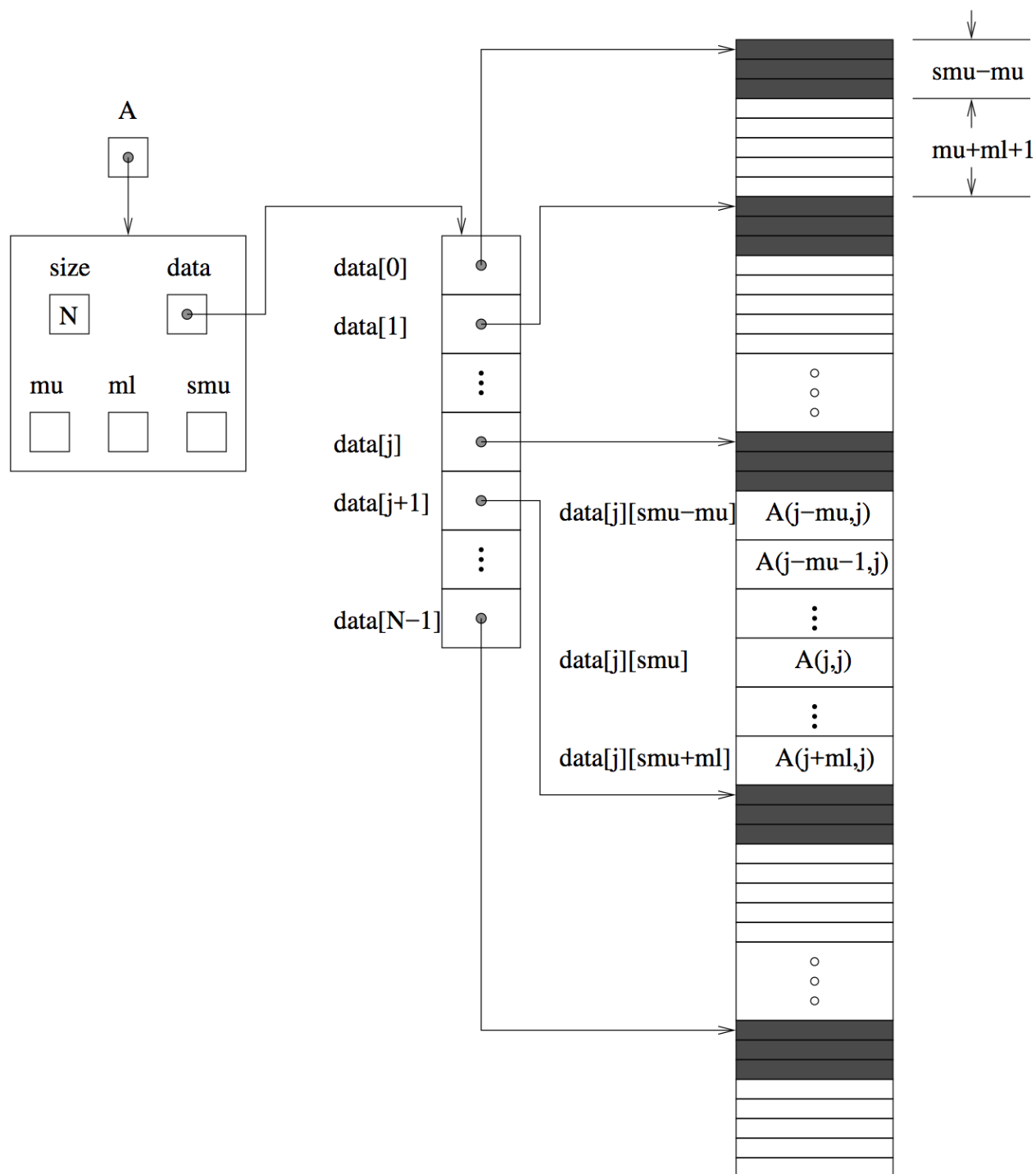


Fig. 9.1: Diagram of the storage for the `SUNMATRIX_BAND` module. Here  $A$  is an  $N \times N$  band matrix with upper and lower half-bandwidths  $\mu$  and  $m_l$ , respectively. The rows and columns of  $A$  are numbered from 0 to  $N-1$  and the  $(i, j)$ -th element of  $A$  is denoted  $A(i, j)$ . The greyed out areas of the underlying component storage are used by the associated `SUNLINSOL_BAND` or `SUNLINSOL_LAPACKBAND` linear solver.

**SM\_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 )
```

**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`,  $0 \leq j \leq N - 1$ . The type of the expression `SM_COLUMN_B(A, j)` is `sunrealtype *`. 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  $0 \leq i, j \leq N - 1$ . The location  $(i, j)$  should further satisfy  $j - \text{mu} \leq i \leq j + \text{ml}$ .

Implementation:

```
#define SM_ELEMENT_B(A, i, j)    ( (SM_CONTENT_B(A)->cols)[j][(i)-(j)+SM_SUBAND_B(A)] )
```

### SM\_COLUMN\_ELEMENT\_B(A)

This macro gives access to the individual entries of the data array of a banded SUNMatrix.

The assignments `SM_COLUMN_ELEMENT_B(col_j, i, j) = a_ij` and `a_ij = SM_COLUMN_ELEMENT_B(col_j, i, j)` reference the  $(i, j)$ -th entry of the band matrix `A` when used in conjunction with `SM_COLUMN_B` to reference the  $j$ -th column through `col_j`. The index  $(i, j)$  should satisfy  $j - \text{mu} \leq i \leq j + \text{ml}$ .

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 §9.2. 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, *SUNContext* sunctx)

This constructor function creates and allocates memory for a banded `SUNMatrix`. Its arguments are the matrix size, N, and the upper and lower half-bandwidths of the matrix, mu and ml. The stored upper bandwidth is set to mu+ml to accommodate subsequent factorization in the `SUNLINSOL_BAND` and `SUNLINSOL_LAPACK-BAND` modules.

*SUNMatrix* **SUNBandMatrixStorage**(*sunindextype* N, *sunindextype* mu, *sunindextype* ml, *sunindextype* smu, *SUNContext* sunctx)

This constructor function creates and allocates memory for a banded `SUNMatrix`. Its arguments are the matrix size, N, the upper and lower half-bandwidths of the matrix, mu and ml, and the stored upper bandwidth, smu. When creating a band `SUNMatrix`, this value should be

- at least  $\min(N-1, \mu+\text{ml})$  if the matrix will be used by the `SUNLinSol_Band` module;
- exactly equal to  $\mu+\text{ml}$  if the matrix will be used by the `SUNLinSol_LapackBand` module;
- at least  $\mu$  if used in some other manner.

#### Note

It is strongly recommended that users call the default constructor, `SUNBandMatrix()`, in all standard use cases. This advanced constructor is used internally within SUNDIALS solvers, and is provided to users who require banded matrices for non-default purposes.

**void SUNBandMatrix\_Print**(*SUNMatrix* A, FILE \*outfile)

This function prints the content of a banded `SUNMatrix` to the output stream specified by outfile. Note: `stdout` or `stderr` may be used as arguments for outfile to print directly to standard output or standard error, respectively.

*sunindextype* **SUNBandMatrix\_Rows**(*SUNMatrix* A)

This function returns the number of rows in the banded `SUNMatrix`.

*sunindextype* **SUNBandMatrix\_Columns**(*SUNMatrix* A)

This function returns the number of columns in the banded `SUNMatrix`.

*sunindextype* **SUNBandMatrix\_LowerBandwidth**(*SUNMatrix* A)

This function returns the lower half-bandwidth for the banded `SUNMatrix`.

*sunindextype* **SUNBandMatrix\_UpperBandwidth**(*SUNMatrix* A)

This function returns the upper half-bandwidth of the banded `SUNMatrix`.

*sunindextype* **SUNBandMatrix\_StoredUpperBandwidth**(*SUNMatrix* A)

This function returns the stored upper half-bandwidth of the banded `SUNMatrix`.

*sunindextype* **SUNBandMatrix\_LDim**(*SUNMatrix* A)

This function returns the length of the leading dimension of the banded `SUNMatrix`.

*sunindextype* **SUNBandMatrix\_LData**(*SUNMatrix* A)

This function returns the length of the data array for the banded `SUNMatrix`.

*sunrealtype* \***SUNBandMatrix\_Data**(*SUNMatrix* A)

This function returns a pointer to the data array for the banded `SUNMatrix`.

*sunrealtype* \*\***SUNBandMatrix\_Cols**(*SUNMatrix* A)

This function returns a pointer to the cols array for the band *SUNMatrix*.

*sunrealtype* \***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  $m1$ .

#### Warning

When calling this function from the Fortran interfaces the shape of the array that is returned is [1], and the only element you can (legally) access is the diagonal element. Fortran users should instead work with the data array returned by [SUNBandMatrix\\_Data\(\)](#) directly.

#### 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 = SUNBandMatrix_Data(A)`, or equivalently `A_data = SM_DATA_B(A)`, and then access `A_data[i]` within the loop.
  - First obtain the array of column pointers via `A_cols = SUNBandMatrix_Cols(A)`, or equivalently `A_cols = SM_COLS_B(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.

## 9.7 The SUNMATRIX\_CUSPARSE Module

The `SUNMATRIX_CUSPARSE` module is an interface to the NVIDIA cuSPARSE matrix for use on NVIDIA GPUs [7]. All data stored by this matrix implementation resides on the GPU at all times.

The header file to be included when using this module is `sunmatrix/sunmatrix_cusparse.h`. The installed library to link to is `libsundials_sunmatrixcusparse.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

### 9.7.1 SUNMATRIX\_CUSPARSE Description

The implementation currently supports the cuSPARSE CSR matrix format described in the cuSPARSE documentation, as well as a unique low-storage format for block-diagonal matrices of the form

$$A = \begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix},$$

where all the block matrices  $\mathbf{A}_j$  share the same sparsity pattern. We will refer to this format as BCSR (not to be confused with the canonical BSR format where each block is stored as dense). In this format, the CSR column indices



and row pointers are only stored for the first block and are computed only as necessary for other blocks. This can drastically reduce the amount of storage required compared to the regular CSR format when the number of blocks is large. This format is well-suited for, and intended to be used with, the `SUNLinearSolver_cuSolverSp_batchQR` linear solver (see §10.17).

**The `SUNMATRIX_CUSPARSE` module is experimental and subject to change.**

### 9.7.2 `SUNMATRIX_CUSPARSE` Functions

The `SUNMATRIX_CUSPARSE` module defines GPU-enabled sparse implementations of all matrix operations listed in §9.2 except for the `SUNMatSpace()` and `SUNMatMatvecSetup()` operations:

- `SUNMatGetID_cuSparse` – returns `SUNMATRIX_CUSPARSE`
- `SUNMatClone_cuSparse`
- `SUNMatDestroy_cuSparse`
- `SUNMatZero_cuSparse`
- `SUNMatCopy_cuSparse`
- `SUNMatScaleAdd_cuSparse` – performs  $A = cA + B$ , where  $A$  and  $B$  must have the same sparsity pattern
- `SUNMatScaleAddI_cuSparse` – performs  $A = cA + I$ , where the diagonal of  $A$  must be present
- `SUNMatMatvec_cuSparse`

In addition, the `SUNMATRIX_CUSPARSE` module defines the following implementation specific functions:

*SUNMatrix* **`SUNMatrix_cuSparse_NewCSR`**(int M, int N, int NNZ, `cusparseHandle_t` cusp, *SUNContext* sunctx)

This constructor function creates and allocates memory for a `SUNMATRIX_CUSPARSE` `SUNMatrix` that uses the CSR storage format. Its arguments are the number of rows and columns of the matrix, M and N, the number of nonzeros to be stored in the matrix, NNZ, and a valid `cusparseHandle_t`.

*SUNMatrix* **`SUNMatrix_cuSparse_NewBlockCSR`**(int nblocks, int blockrows, int blockcols, int blocknnz, `cusparseHandle_t` cusp, *SUNContext* sunctx)

This constructor function creates and allocates memory for a `SUNMATRIX_CUSPARSE` `SUNMatrix` object that leverages the `SUNMAT_CUSPARSE_BCSR` storage format to store a block diagonal matrix where each block shares the same sparsity pattern. The blocks must be square. The function arguments are the number of blocks, nblocks, the number of rows, blockrows, the number of columns, blockcols, the number of nonzeros in each block, blocknnz, and a valid `cusparseHandle_t`.

#### Warning

The `SUNMAT_CUSPARSE_BCSR` format currently only supports square matrices, i.e., `blockrows == blockcols`.

*SUNMatrix* **`SUNMatrix_cuSparse_MakeCSR`**(`cusparseMatDescr_t` mat\_descr, int M, int N, int NNZ, int \*rowptrs, int \*colind, *sunrealtype* \*data, `cusparseHandle_t` cusp, *SUNContext* sunctx)

This constructor function creates a `SUNMATRIX_CUSPARSE` `SUNMatrix` object from user provided pointers. Its arguments are a `cusparseMatDescr_t` that must have index base `CUSPARSE_INDEX_BASE_ZERO`, the number of rows and columns of the matrix, M and N, the number of nonzeros to be stored in the matrix, NNZ, and a valid `cusparseHandle_t`.

int **SUNMatrix\_cuSparse\_Rows**(*SUNMatrix* A)

This function returns the number of rows in the sparse *SUNMatrix*.

int **SUNMatrix\_cuSparse\_Columns**(*SUNMatrix* A)

This function returns the number of columns in the sparse *SUNMatrix*.

int **SUNMatrix\_cuSparse\_NNZ**(*SUNMatrix* A)

This function returns the number of entries allocated for nonzero storage for the sparse *SUNMatrix*.

int **SUNMatrix\_cuSparse\_SparseType**(*SUNMatrix* A)

This function returns the storage type (*SUNMAT\_CUSPARSE\_CSR* or *SUNMAT\_CUSPARSE\_BCSR*) for the sparse *SUNMatrix*.

*sunrealtype* \***SUNMatrix\_cuSparse\_Data**(*SUNMatrix* A)

This function returns a pointer to the data array for the sparse *SUNMatrix*.

int \***SUNMatrix\_cuSparse\_IndexValues**(*SUNMatrix* A)

This function returns a pointer to the index value array for the sparse *SUNMatrix* – for the CSR format this is an array of column indices for each nonzero entry. For the BCSR format this is an array of the column indices for each nonzero entry in the first block only.

int \***SUNMatrix\_cuSparse\_IndexPointers**(*SUNMatrix* A)

This function returns a pointer to the index pointer array for the sparse *SUNMatrix* – for the CSR format this is an array of the locations of the first entry of each row in the data and *indexvalues* arrays, for the BCSR format this is an array of the locations of each row in the data and *indexvalues* arrays in the first block only.

int **SUNMatrix\_cuSparse\_NumBlocks**(*SUNMatrix* A)

This function returns the number of matrix blocks.

int **SUNMatrix\_cuSparse\_BlockRows**(*SUNMatrix* A)

This function returns the number of rows in a matrix block.

int **SUNMatrix\_cuSparse\_BlockColumns**(*SUNMatrix* A)

This function returns the number of columns in a matrix block.

int **SUNMatrix\_cuSparse\_BlockNNZ**(*SUNMatrix* A)

This function returns the number of nonzeros in each matrix block.

*sunrealtype* \***SUNMatrix\_cuSparse\_BlockData**(*SUNMatrix* A, int blockidx)

This function returns a pointer to the location in the data array where the data for the block, *blockidx*, begins. Thus, *blockidx* must be less than *SUNMatrix\_cuSparse\_NumBlocks*(A). The first block in the *SUNMatrix* is index 0, the second block is index 1, and so on.

cusparseMatDescr\_t **SUNMatrix\_cuSparse\_MatDescr**(*SUNMatrix* A)

This function returns the *cusparseMatDescr\_t* object associated with the matrix.

*SUNErrCode* **SUNMatrix\_cuSparse\_CopyToDevice**(*SUNMatrix* A, *sunrealtype* \*h\_data, int \*h\_idxptrs, int \*h\_idxvals)

This functions copies the matrix information to the GPU device from the provided host arrays. A user may provide NULL for any of *h\_data*, *h\_idxptrs*, or *h\_idxvals* to avoid copying that information.

The function returns *SUN\_SUCCESS* if the copy operation(s) were successful, or a nonzero error code otherwise.

*SUNErrCode* **SUNMatrix\_cuSparse\_CopyFromDevice**(*SUNMatrix* A, *sunrealtype* \*h\_data, int \*h\_idxptrs, int \*h\_idxvals)

This functions copies the matrix information from the GPU device to the provided host arrays. A user may provide NULL for any of *h\_data*, *h\_idxptrs*, or *h\_idxvals* to avoid copying that information. Otherwise:

- The `h_data` array must be at least `SUNMatrix_cuSparse_NNZ(A)*sizeof(sunrealtype)` bytes.
- The `h_idxptrs` array must be at least `(SUNMatrix_cuSparse_BlockDim(A)+1)*sizeof(int)` bytes.
- The `h_idxvals` array must be at least `(SUNMatrix_cuSparse_BlockNNZ(A))*sizeof(int)` bytes.

The function returns `SUN_SUCCESS` if the copy operation(s) were successful, or a nonzero error code otherwise.

*SUNErrCode* **SUNMatrix\_cuSparse\_SetFixedPattern**(*SUNMatrix* A, *sunbooleantype* yesno)

This function changes the behavior of the `SUNMatZero` operation on the object A. By default the matrix sparsity pattern is not considered to be fixed, thus, the `SUNMatZero` operation zeros out all data array as well as the `indexvalues` and `indexpointers` arrays. Providing a value of 1 or `SUNTRUE` for the `yesno` argument changes the behavior of `SUNMatZero` on A so that only the data is zeroed out, but not the `indexvalues` or `indexpointers` arrays. Providing a value of 0 or `SUNFALSE` for the `yesno` argument is equivalent to the default behavior.

*SUNErrCode* **SUNMatrix\_cuSparse\_SetKernelExecPolicy**(*SUNMatrix* A, *SUNCudaExecPolicy* \*exec\_policy)

This function sets the execution policies which control the kernel parameters utilized when launching the CUDA kernels. By default the matrix is setup to use a policy which tries to leverage the structure of the matrix. See §8.10.2 for more information about the *SUNCudaExecPolicy* class.

### 9.7.3 SUNMATRIX\_CUSPARSE Usage Notes

The `SUNMATRIX_CUSPARSE` module only supports 32-bit indexing, thus `SUNDIALS` must be built for 32-bit indexing to use this module.

The `SUNMATRIX_CUSPARSE` module can be used with CUDA streams by calling the `cuSPARSE` function `cusparseSetStream` on the `cusparseHandle_t` that is provided to the `SUNMATRIX_CUSPARSE` constructor.

#### Warning

When using the `SUNMATRIX_CUSPARSE` module with a `SUNDIALS` package (e.g. `ARKODE`), the stream given to `cuSPARSE` should be the same stream used for the `NVECTOR` object that is provided to the package, and the `NVECTOR` object given to the `SUNMatvec` operation. If different streams are utilized, synchronization issues may occur.

## 9.8 The SUNMATRIX\_SPARSE Module

The sparse implementation of the `SUNMatrix` module, `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;
    sunrealtype *data;
    int sparsetype;
    sunindextype *indexvals;
    sunindextype *indexptrs;
    /* CSC indices */
}
```

(continues on next page)

(continued from previous page)

```

sunindextype **rowvals;
sunindextype **colptrs;
/* CSR indices */
sunindextype **colvals;
sunindextype **rowptrs;
};

```

A diagram of the underlying data representation in a sparse matrix is shown in [Fig. 9.2](#). 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 **sunrealtype** variables (of length **NNZ**), containing the values of the nonzero entries in the matrix
- **sparsetype** - type of the sparse matrix (**SUN\_CSC\_MAT** or **SUN\_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 **SUN\_CSC\_MAT**, otherwise set to **NULL**.
- **colptrs** - pointer to **indexptrs** when **sparsetype** is **SUN\_CSC\_MAT**, otherwise set to **NULL**.
- **colvals** - pointer to **indexvals** when **sparsetype** is **SUN\_CSR\_MAT**, otherwise set to **NULL**.
- **rowptrs** - pointer to **indexptrs** when **sparsetype** is **SUN\_CSR\_MAT**, otherwise set to **NULL**.

For example, the  $5 \times 4$  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;

```

(continues on next page)

(continued from previous page)

```
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = SUN_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 = SUN_CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with \* may contain any values). Note in both cases that the final value in `indexptrs` is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
NP = M;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = SUN_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:

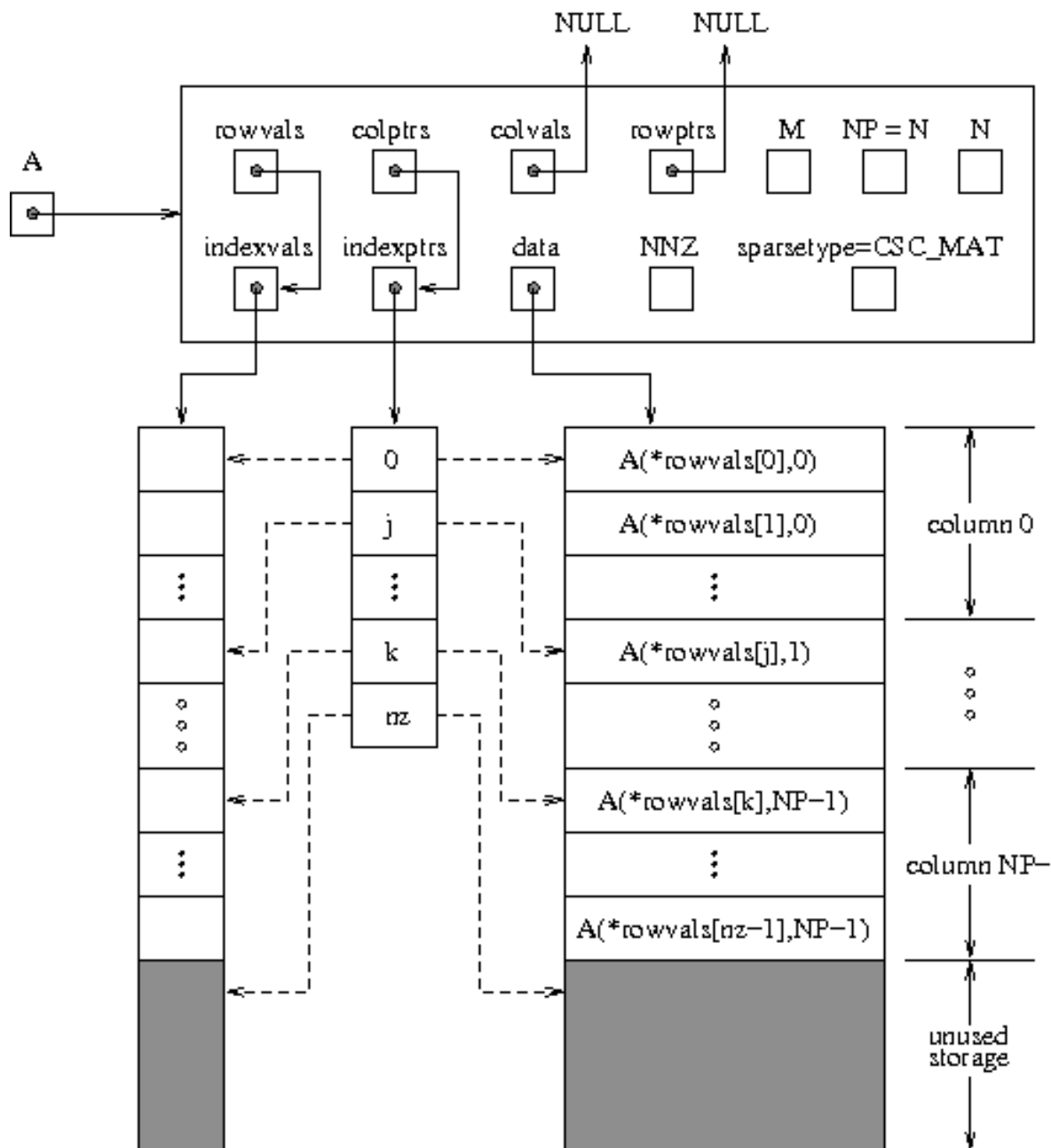


Fig. 9.2: Diagram of the storage for a compressed-sparse-column matrix of type `SUNMATRIX_SPARSE`: Here  $A$  is an  $M \times N$  sparse CSC matrix with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 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 §9.2. 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, *SUNContext* sunctx)

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 `SUN_CSR_MAT` or `SUN_CSC_MAT`).

*SUNMatrix* **SUNSparseFromDenseMatrix**(*SUNMatrix* A, *sunrealtype* 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 `SUN_CSC_MAT` or `SUN_CSR_MAT`

The function returns `NULL` if any requirements are violated, or if the matrix storage request cannot be satisfied.

*SUNMatrix* **SUNSparseFromBandMatrix**(*SUNMatrix* A, *sunrealtype* 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 `SUN_CSC_MAT` or `SUN_CSR_MAT`.

The function returns `NULL` if any requirements are violated, or if the matrix storage request cannot be satisfied.

*SUNErrCode* **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 a *SUNErrCode*.

*SUNErrCode* **SUNSparseMatrix\_Reallocate**(*SUNMatrix* A, *sunindextype* NNZ)

Function to reallocate internal sparse matrix storage arrays so that the resulting sparse matrix has storage for a specified number of nonzeros. Returns a *SUNErrCode*.



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 (`SUN_CSR_MAT` or `SUN_CSC_MAT`) for the sparse *SUNMatrix*.

*sunrealtype* \***SUNsparseMatrix\_Data**(*SUNMatrix* A)

This function returns a pointer to the data array for the sparse *SUNMatrix*.

*sunindextype* \***SUNsparseMatrix\_IndexValues**(*SUNMatrix* A)

This function returns a pointer to index value array for the sparse *SUNMatrix* – for CSR format this is the column index for each nonzero entry, for CSC format this is the row index for each nonzero entry.

*sunindextype* \***SUNsparseMatrix\_IndexPointers**(*SUNMatrix* A)

This function returns a pointer to the index pointer array for the sparse *SUNMatrix* – for CSR format this is the location of the first entry of each row in the data and `indexvalues` arrays, for CSC format this is the location of the first entry of each column.

#### Note

Within the `SUNMatMatvec_Sparse` routine, internal consistency checks are performed to ensure that the matrix is called with consistent `N_Vector` implementations. These are currently limited to: `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, `NVECTOR_PTHREADS`, and `NVECTOR_CUDA` when using managed memory. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

## 9.9 The SUNMATRIX\_SLUNRLOC Module

The `SUNMATRIX_SLUNRLOC` module is an interface to the `SuperMatrix` structure provided by the `SuperLU_DIST` sparse matrix factorization and solver library written by X. Sherry Li and collaborators [8, 51, 77, 78]. It is designed to be used with the `SuperLU_DIST` `SUNLinearSolver` module discussed in §10.15. To this end, it defines the content field of *SUNMatrix* to be the following structure:

```
struct _SUNMatrixContent_SLUNRloc {
    sunboolean_t own_data;
    gridinfo_t   *grid;
    sunindextype *row_to_proc;
    pdgsmv_comm_t *gsmv_comm;
    SuperMatrix   *A_super;
```

(continues on next page)

(continued from previous page)

```
SuperMatrix *ACS_super;
};
```

A more complete description of the this content field is given below:

- `own_data` – a flag which indicates if the `SUNMatrix` is responsible for freeing `A_super`
- `grid` – pointer to the `SuperLU_DIST` structure that stores the 2D process grid
- `row_to_proc` – a mapping between the rows in the matrix and the process it resides on; will be `NULL` until the `SUNMatMatvecSetup` routine is called
- `gsmv_comm` – pointer to the `SuperLU_DIST` structure that stores the communication information needed for matrix-vector multiplication; will be `NULL` until the `SUNMatMatvecSetup` routine is called
- `A_super` – pointer to the underlying `SuperLU_DIST` `SuperMatrix` with `Stype = SLU_NR_loc`, `Dtype = SLU_D`, `Mtype = SLU_GE`; must have the full diagonal present to be used with `SUNMatScaleAddI` routine
- `ACS_super` – a column-sorted version of the matrix needed to perform matrix-vector multiplication; will be `NULL` until the routine `SUNMatMatvecSetup` routine is called

The header file to include when using this module is `sunmatrix/sunmatrix_slunrloc.h`. The installed module library to link to is `libsundials_sunmatrixslunrloc.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

## 9.9.1 SUNMATRIX\_SLUNRLOC Functions

The `SUNMATRIX_SLUNRLOC` module provides the following user-callable routines:

*SUNMatrix* **SUNMatrix\_SLUNRloc**(`SuperMatrix *Asuper`, `gridinfo_t *grid`, *SUNContext* `sunctx`)

This constructor function creates and allocates memory for a `SUNMATRIX_SLUNRLOC` object. Its arguments are a fully-allocated `SuperLU_DIST` `SuperMatrix` with `Stype = SLU_NR_loc`, `Dtype = SLU_D`, `Mtype = SLU_GE` and an initialized `SuperLU_DIST` 2D process grid structure. It returns a `SUNMatrix` object if `Asuper` is compatible else it returns `NULL`.

void **SUNMatrix\_SLUNRloc\_Print**(*SUNMatrix* `A`, `FILE *fp`)

This function prints the underlying `SuperMatrix` content. It is useful for debugging. Its arguments are the `SUNMatrix` object and a `FILE` pointer to print to. It returns void.

`SuperMatrix *`**SUNMatrix\_SLUNRloc\_SuperMatrix**(*SUNMatrix* `A`)

This function returns the underlying `SuperMatrix` of `A`. Its only argument is the `SUNMatrix` object to access.

`gridinfo_t *`**SUNMatrix\_SLUNRloc\_ProcessGrid**(*SUNMatrix* `A`)

This function returns the `SuperLU_DIST` 2D process grid associated with `A`. Its only argument is the `SUNMatrix` object to access.

*sunbooleantype* **SUNMatrix\_SLUNRloc\_OwnData**(*SUNMatrix* `A`)

This function returns true if the `SUNMatrix` object is responsible for freeing the underlying `SuperMatrix`, otherwise it returns false. Its only argument is the `SUNMatrix` object to access.

The `SUNMATRIX_SLUNRLOC` module also defines implementations of all generic `SUNMatrix` operations listed in §9.2:

- `SUNMatGetID_SLUNRloc` – returns `SUNMATRIX_SLUNRLOC`
- `SUNMatClone_SLUNRloc`
- `SUNMatDestroy_SLUNRloc`

- `SUNMatSpace_SLUNRloc` – this only returns information for the storage within the matrix interface, i.e. storage for `row_to_proc`
- `SUNMatZero_SLUNRloc`
- `SUNMatCopy_SLUNRloc`
- `SUNMatScaleAdd_SLUNRloc` – performs  $A = cA + B$ , where  $A$  and  $B$  must have the same sparsity pattern
- `SUNMatScaleAddI_SLUNRloc` – performs  $A = cA + I$ , where the diagonal of  $A$  must be present
- `SUNMatMatvecSetup_SLUNRloc` – initializes the SuperLU\_DIST parallel communication structures needed to perform a matrix-vector product; only needs to be called before the first call to `SUNMatMatvec()` or if the matrix changed since the last setup
- `SUNMatMatvec_SLUNRloc`

## 9.10 The SUNMATRIX\_GINKGO Module

Added in version 6.4.0.

The `SUNMATRIX_GINKGO` implementation of the `SUNMatrix` API provides an interface to the matrix data structure for the Ginkgo linear algebra library [11]. Ginkgo provides several different matrix formats and linear solvers which can run on a variety of hardware, such as NVIDIA, AMD, and Intel GPUs as well as multicore CPUs. Since Ginkgo is a modern C++ library, `SUNMATRIX_GINKGO` is also written in modern C++ (it requires C++14). Unlike most other SUNDIALS modules, it is a header only library. To use the `SUNMATRIX_GINKGO` `SUNMatrix`, users will need to include `sunmatrix/sunmatrix_ginkgo.hpp`. More instructions on building SUNDIALS with Ginkgo enabled are given in §17.3.19. For instructions on building and using Ginkgo itself, refer to the [Ginkgo website and documentation](#).

### Note

It is assumed that users of this module are aware of how to use Ginkgo. This module does not try to encapsulate Ginkgo matrices, rather it provides a lightweight interoperability layer between Ginkgo and SUNDIALS.

The `SUNMATRIX_GINKGO` module is defined by the `sundials::ginkgo::Matrix` templated class:

```
template<typename GkoMatType>
class Matrix : public sundials::impl::BaseMatrix, public sundials::ConvertibleTo<SUNMatrix>;
```

### 9.10.1 Compatible Vectors

The `N_Vector` to use with the `SUNLINEARSOLVER_GINKGO` module depends on the `gko::Executor` utilized. That is, when using the `gko::CudaExecutor` you should use a CUDA capable `N_Vector` (e.g., §8.10), `gko::HipExecutor` goes with a HIP capable `N_Vector` (e.g., §8.11), `gko::DpcppExecutor` goes with a DPC++/SYCL capable `N_Vector` (e.g., §8.12), and `gko::OmpExecutor` goes with a CPU based `N_Vector` (e.g., §8.6). Specifically, what makes a `N_Vector` compatible with different Ginkgo executors is where they store the data. The GPU enabled Ginkgo executors need the data to reside on the GPU, so the `N_Vector` must implement `N_VGetDeviceArrayPointer()` and keep the data in GPU memory. The CPU-only enabled Ginkgo executors (e.g. `gko::OmpExecutor` and `gko::ReferenceExecutor`) need data to reside on the CPU and will use `N_VGetArrayPointer()` to access the `N_Vector` data.

### 9.10.2 Using SUNMATRIX\_GINKGO

To use the SUNMATRIX\_GINKGO module, we begin by creating an instance of a Ginkgo matrix using Ginkgo's API. For example, below we create a Ginkgo sparse matrix that uses the CSR storage format and then fill the diagonal of the matrix with ones to make an identity matrix:

```
auto gko_matrix{gko::matrix::Csr<sunrealtype, sunindextype>::create(gko_exec, matrix_dim)};
gko_matrix->read(gko::matrix_data<sunrealtype, sunindextype>::diag(matrix_dim, 1.0));
```

After we have a Ginkgo matrix object, we wrap it in an instance of the `sundials::ginkgo::Matrix` class. This object can be provided to other SUNDIALS functions that expect a `SUNMatrix` object via implicit conversion, or the `get()` method:

```
sundials::ginkgo::Matrix<gko::matrix::Csr> matrix{gko_matrix, sunctx};
SUNMatrix I1 = matrix.get(); // explicit conversion to SUNMatrix
SUNMatrix I2 = matrix;      // implicit conversion to SUNMatrix
```

No further interaction with `matrix` is required from this point, and it is possible to use the `SUNMatrix` API operating on `I1` or `I2` (or if needed, via Ginkgo operations on `gko_matrix`).

#### Warning

`SUNMatDestroy()` should never be called on a `SUNMatrix` that was created via conversion from a `sundials::ginkgo::Matrix`. Doing so may result in a double free.

### 9.10.3 SUNMATRIX\_GINKGO API

In this section we list the public API of the `sundials::ginkgo::Matrix` class.

```
template<typename GkoMatType>
class Matrix : public sundials::impl::BaseMatrix, public sundials::ConvertibleTo<SUNMatrix>
```

**Matrix()** = default

Default constructor - means the matrix must be copied or moved to.

**Matrix**(std::shared\_ptr<*GkoMatType*> gko\_mat, SUNContext sunctx)

Constructs a `Matrix` from an existing Ginkgo matrix object.

#### Parameters

- **gko\_mat** – A Ginkgo matrix object
- **sunctx** – The SUNDIALS simulation context object (*SUNContext*)

**Matrix**(*Matrix* &&that\_matrix) noexcept

Move constructor.

**Matrix**(const *Matrix* &that\_matrix)

Copy constructor (performs a deep copy).

*Matrix* &operator=(*Matrix* &&rhs) noexcept

Move assignment.

*Matrix* &operator=(const *Matrix* &rhs)

Copy assignment clones the `gko::matrix` and *SUNMatrix*. This is a deep copy (i.e. a new data array is created).

virtual **~Matrix()** = default;

Default destructor.

std::shared\_ptr<*GkoMatType*> **GkoMtx()** const

Get the underlying Ginkgo matrix object.

std::shared\_ptr<const gko::Executor> **GkoExec()** const

Get the `gko::Executor` associated with the Ginkgo matrix.

const gko::dim<2> &**GkoSize()** const

Get the size, i.e. `gko::dim`, for the Ginkgo matrix.

**operator SUNMatrix()** override

Implicit conversion to a *SUNMatrix*.

**operator SUNMatrix()** const override

Implicit conversion to a *SUNMatrix*.

*SUNMatrix* **get()** override

Explicit conversion to a *SUNMatrix*.

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

*SUNMatrix* **get()** const override

Explicit conversion to a *SUNMatrix*.

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

## 9.11 The SUNMATRIX\_GINKGOBATCH Module

Added in version 7.5.0.

The `SUNMATRIX_GINKGOBATCH` implementation of the *SUNMatrix* API provides an interface to the batched matrix types from the Ginkgo linear algebra library. This module is written in C++17 and is distributed as a header file. To use the `SUNMATRIX_GINKGOBATCH` *SUNMatrix*, users will need to include `sunmatrix/sunmatrix_ginkgobatch.hpp`. The module is meant to be used with the `SUNLINEARSOLVER_GINKGOBATCH` module described in §10.19.

### Note

It is assumed that users of this module are aware of how to use Ginkgo. This module does not try to encapsulate Ginkgo matrices, rather it provides a lightweight interoperability layer between Ginkgo and SUNDIALS. Most, if not all, of the Ginkgo batch matrix types should work with this interface.

### 9.11.1 Compatible Vectors

The *N\_Vector* to use with the `SUNLINEARSOLVER_GINKGOBATCH` module depends on the `gko::Executor` utilized. That is, when using the `gko::CudaExecutor` you should use a CUDA capable *N\_Vector* (e.g., §8.10), `gko::HipExecutor` goes with a HIP capable *N\_Vector* (e.g., §8.11), `gko::DpcppExecutor` goes with a DPC++/SYCL capable *N\_Vector* (e.g., §8.12), and `gko::OmpExecutor` goes with a CPU based *N\_Vector* (e.g., §8.6). Specifically, what makes a *N\_Vector* compatible with different Ginkgo executors is where they store the data. The GPU enabled Ginkgo executors need the data to reside on the GPU, so the *N\_Vector* must implement *N\_VGetDeviceArrayPointer()* and keep the data in GPU memory. The CPU-only enabled Ginkgo executors (e.g. `gko::OmpExecutor`

and `gko::ReferenceExecutor`) need data to reside on the CPU and will use `N_VGetArrayPointer()` to access the `N_Vector` data.

### 9.11.2 Compatible Packages

This module will work with any of the SUNDIALS packages. The only caveat is that, when using ARKODE with a non-identity mass matrix, the only Ginkgo matrix type currently supported is `BatchDense`.

### 9.11.3 SUNMATRIX\_GINKGOBATCH API

In this section we list the public API of the `sundials::ginkgo::BatchMatrix` class.

```
template<class GkoBatchMatType>
class sundials::ginkgo::BatchMatrix : public sundials::ConvertibleTo<SUNMatrix>
    Batched matrix wrapper for Ginkgo batch matrix types, providing a SUNDIALS SUNMatrix interface.

    BatchMatrix()
        Default constructor. The matrix must be copied or moved to.

    BatchMatrix(gko::size_type num_batches, sunindextype M, sunindextype N, std::shared_ptr<const
        gko::Executor> gko_exec, SUNContext sunctx)
        Construct a batch matrix with the given number of batches, rows M, columns N, executor, and context.
        (Specialized for supported Ginkgo batch matrix types.)

    BatchMatrix(gko::size_type num_batches, sunindextype M, sunindextype N, sunindextype num_nonzeros,
        std::shared_ptr<const gko::Executor> gko_exec, SUNContext sunctx)
        Construct a batch sparse matrix with the given number of batches, rows M, columns N, nonzeros, executor,
        and context. (Specialized for supported Ginkgo batch matrix types.)

    BatchMatrix(std::shared_ptr<GkoBatchMatType> gko_mat, SUNContext sunctx)
        Construct a BatchMatrix from an existing Ginkgo batch matrix pointer and SUNDIALS context.

    BatchMatrix(BatchMatrix &&that_matrix) noexcept
        Move constructor.

    BatchMatrix(const BatchMatrix &that_matrix)
        Copy constructor. Clones the Ginkgo matrix and SUNDIALS SUNMatrix.

    BatchMatrix &operator=(BatchMatrix &&rhs) noexcept
        Move assignment.

    BatchMatrix &operator=(const BatchMatrix &rhs)
        Copy assignment. Clones the Ginkgo matrix and SUNDIALS SUNMatrix.

    ~BatchMatrix() override = default
        Default destructor.

    std::shared_ptr<GkoBatchMatType> GkoMtx() const
        Get the underlying Ginkgo batch matrix pointer.

    std::shared_ptr<const gko::Executor> GkoExec() const
        Get the Ginkgo executor associated with the matrix.

    const gko::batch_dim<2> &GkoSize() const
        Get the Ginkgo batch size object.
```

sunindextype **NumBatches**() const

Get the number of batches (batch systems).

**operator SUNMatrix**() override

Implicit conversion to a *SUNMatrix*.

**operator SUNMatrix**() const override

Implicit conversion to a *SUNMatrix*.

*SUNMatrix* **get**() override

Explicit conversion to a *SUNMatrix*.

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

*SUNMatrix* **get**() const override

Explicit conversion to a *SUNMatrix*.

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

## 9.12 The SUNMATRIX\_KOKKOSDENSE Module

Added in version 6.4.0.

The SUNMATRIX\_KOKKOSDENSE *SUNMatrix* implementation provides a data structure for dense and dense batched (block-diagonal) matrices using Kokkos [39, 119] and KokkosKernels [118] to support a variety of back-ends including serial, OpenMP, CUDA, HIP, and SYCL. Since Kokkos is a modern C++ library, the module is also written in modern C++ (it requires C++14) as a header only library. To utilize this *SUNMatrix* users will need to include `sunmatrix/sunmatrix_kokkosdense.hpp`. More instructions on building SUNDIALS with Kokkos and KokkosKernels enabled are given in §17.3.24. For instructions on building and using Kokkos and KokkosKernels, refer to the [Kokkos](#) and [KokkosKernels](#) documentation.

### 9.12.1 Using SUNMATRIX\_KOKKOSDENSE

The SUNMATRIX\_KOKKOSDENSE module is defined by the `DenseMatrix` templated class in the `sundials::kokkos` namespace:

```
template<class ExecutionSpace = Kokkos::DefaultExecutionSpace,
         class MemorySpace = typename ExecutionSpace::memory_space>
class DenseMatrix : public sundials::impl::BaseMatrix,
                   public sundials::ConvertibleTo<SUNMatrix>
```

To use the SUNMATRIX\_KOKKOSDENSE module, we begin by constructing an instance of the Kokkos dense matrix e.g.,

```
// Single matrix using the default execution space
sundials::kokkos::DenseMatrix<> A{rows, cols, sunctx};

// Batched (block-diagonal) matrix using the default execution space
sundials::kokkos::DenseMatrix<> Abatch{blocks, rows, cols, sunctx};

// Batched (block-diagonal) matrix using the Cuda execution space
sundials::kokkos::DenseMatrix<Kokkos::Cuda> Abatch{blocks, rows, cols, sunctx};
```

(continues on next page)



(continued from previous page)

```
// Batched (block-diagonal) matrix using the Cuda execution space and
// a non-default execution space instance
sundials::kokkos::DenseMatrix<Kokkos::Cuda> Abatch{blocks, rows, cols,
                                                exec_space_instance,
                                                sunctx};
```

Instances of the `DenseMatrix` class are implicitly or explicitly (using the `get()` method) convertible to a `SUNMatrix` e.g.,

```
sundials::kokkos::DenseMatrix<> A{rows, cols, sunctx};
SUNMatrix B = A;                // implicit conversion to SUNMatrix
SUNMatrix C = A.get();           // explicit conversion to SUNMatrix
```

No further interaction with a `DenseMatrix` is required from this point, and it is possible to use the `SUNMatrix` API to operate on B or C.

### Warning

`SUNMatDestroy()` should never be called on a `SUNMatrix` that was created via conversion from a `sundials::kokkos::DenseMatrix`. Doing so may result in a double free.

The underlying `DenseMatrix` can be extracted from a `SUNMatrix` using `GetDenseMat()` e.g.,

```
auto A_dense_mat = GetDenseMat<>(A_sunmat);
```

The `SUNMATRIX_KOKKOSDENSE` module is compatible with the `NVECTOR_KOKKOS` vector module (see §8.14) and `SUNLINEARSOLVER_KOKKOSDENSE` linear solver module (see §10.20).

## 9.12.2 SUNMATRIX\_KOKKOSDENSE API

In this section we list the public API of the `sundials::kokkos::DenseMatrix` class.

```
template<class ExecutionSpace = Kokkos::DefaultExecutionSpace, class MemorySpace = typename
ExecutionSpace::memory_space>
```

```
class DenseMatrix : public sundials::impl::BaseMatrix, public sundials::ConvertibleTo<SUNMatrix>
```

```
    using exec_space = ExecutionSpace;
```

```
    using memory_space = MemorySpace;
```

```
    using view_type = Kokkos::View<sunrealtype***, memory_space>;
```

```
    using size_type = typename view_type::size_type;
```

```
    using range_policy = Kokkos::MDRangePolicy<exec_space, Kokkos::Rank<3>>;
```

```
    using team_policy = typename Kokkos::TeamPolicy<exec_space>;
```

```
    using member_type = typename Kokkos::TeamPolicy<exec_space>::member_type;
```

```
    DenseMatrix() = default
```

Default constructor – the matrix must be copied or moved to.



**DenseMatrix**(*size\_type* rows, *size\_type* cols, SUNContext sunctx)

Constructs a single DenseMatrix using the default execution space instance.

**Parameters**

- **rows** – number of matrix rows
- **cols** – number of matrix columns
- **sunctx** – the SUNDIALS simulation context object (*SUNContext*)

**DenseMatrix**(*size\_type* rows, *size\_type* cols, *exec\_space* ex, SUNContext sunctx)

Constructs a single DenseMatrix using the provided execution space instance.

**Parameters**

- **rows** – number of matrix rows
- **cols** – number of matrix columns
- **ex** – an execution space
- **sunctx** – the SUNDIALS simulation context object (*SUNContext*)

**DenseMatrix**(*size\_type* blocks, *size\_type* block\_rows, *size\_type* block\_cols, SUNContext sunctx)

Constructs a batched (block-diagonal) DenseMatrix using the default execution space instance.

**Parameters**

- **blocks** – number of matrix blocks
- **block\_rows** – number of rows in a block
- **block\_cols** – number of columns in a block
- **sunctx** – the SUNDIALS simulation context object (*SUNContext*)

**DenseMatrix**(*size\_type* blocks, *size\_type* block\_rows, *size\_type* block\_cols, *exec\_space* ex, SUNContext sunctx)

Constructs a batched (block-diagonal) DenseMatrix using the provided execution space instance.

**Parameters**

- **blocks** – number of matrix blocks
- **block\_rows** – number of rows in a block
- **block\_cols** – number of columns in a block
- **ex** – an execution space
- **sunctx** – the SUNDIALS simulation context object (*SUNContext*)

**DenseMatrix**(*DenseMatrix* &that\_matrix) noexcept

Move constructor.

**DenseMatrix**(const *DenseMatrix* &that\_matrix)

Copy constructor. This creates a shallow clone of the Matrix, i.e., it creates a new Matrix with the same properties, such as size, but it does not copy the data.

*DenseMatrix* &operator=(*DenseMatrix* &&rhs) noexcept

Move assignment.

*DenseMatrix* &**operator**=(const *DenseMatrix* &rhs)

Copy assignment. This creates a shallow clone of the Matrix, i.e., it creates a new Matrix with the same properties, such as size, but it does not copy the data.

virtual ~**DenseMatrix**() = default;

Default destructor.

*exec\_space* **ExecSpace**()

Get the execution space instance used by the matrix.

*view\_type* **View**()

Get the underlying Kokkos view with extents {blocks, block\_rows, block\_cols}.

*size\_type* **Blocks**()

Get the number of blocks i.e., extent(0).

*size\_type* **BlockRows**()

Get the number of rows in a block i.e., extent(1).

*size\_type* **BlockCols**()

Get the number of columns in a block i.e., extent(2).

*size\_type* **Rows**()

Get the number of rows in the block-diagonal matrix i.e., extent(0) \* extent(1).

*size\_type* **Cols**()

Get the number of columns in the block-diagonal matrix i.e., extent(0) \* extent(2).

**operator SUNMatrix**() override

Implicit conversion to a *SUNMatrix*.

**operator SUNMatrix**() const override

Implicit conversion to a *SUNMatrix*.

*SUNMatrix* **get**() override

Explicit conversion to a *SUNMatrix*.

Added in version 7.6.0: Replaces the *Convert* method which was deprecated.

*SUNMatrix* **get**() const override

Explicit conversion to a *SUNMatrix*.

Added in version 7.6.0: Replaces the *Convert* method which was deprecated.

template<class **ExecutionSpace** = Kokkos::DefaultExecutionSpace, class **MemorySpace** = typename *ExecutionSpace*::memory\_space>

inline *DenseMatrix*<MatrixType> \***GetDenseMat**(*SUNMatrix* A)

Get the dense matrix wrapped by a *SUNMatrix*

## 9.13 SUNMATRIX Examples

There are *SUNMatrix* examples that may be installed for each implementation, that make 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_SUNMatMatvecSetup`: verifies that `SUNMatMatvecSetup()` can be called.
- `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  $z = Ax$  using `SUNMatMatvec()`, and verifies that  $y = z$ .
- `Test_SUNMatSpace`: verifies that `SUNMatSpace()` can be called, and outputs the results to `stdout`.

## 9.14 SUNMATRIX functions used by ARKODE

In Table Table 9.2, we list the matrix functions in the `SUNMatrix` module used within the ARKODE package. The table also shows, for each function, which of the code modules uses the function. The main ARKODE time step modules, `ARKStep`, `ERKStep`, and `MRISStep`, do not call any `SUNMatrix` functions directly, so the table columns are specific to the `ARKLS` interface and the `ARKBANDPRE` and `ARKBBDPRE` preconditioner modules. We further note that the `ARKLS` interface only utilizes these routines when supplied with a *matrix-based* linear solver, i.e. the `SUNMatrix` object ( $J$  or  $M$ ) passed to `ARKStepSetLinearSolver()` or `ARKStepSetMassLinearSolver()` was not `NULL`.

At this point, we should emphasize that the ARKODE user does not need to know anything about the usage of matrix functions by the ARKODE code modules in order to use ARKODE. The information is presented as an implementation detail for the interested reader.

Table 9.2: List of matrix functions usage by ARKODE code modules

	ARKLS	ARKBANDPRE	ARKBBDPRE
<code>SUNMatGetID()</code>	X		
<code>SUNMatClone()</code>	X		
<code>SUNMatDestroy()</code>	X	X	X
<code>SUNMatZero()</code>	X	X	X
<code>SUNMatCopy()</code>	X	X	X
<code>SUNMatScaleAddI()</code>	X	X	X
<code>SUNMatScaleAdd()</code>	1		
<code>SUNMatMatvec()</code>	1		
<code>SUNMatMatvecSetup()</code>	1,2		
<code>SUNMatSpace()</code>	2	2	2

1. These matrix functions are only used for problems involving a non-identity mass matrix.

2. These matrix functions are optionally used, in that these are only called if they are implemented in the `SUNMatrix` module that is being used (i.e. their function pointers are non-NULL). If not supplied, these modules will assume that the matrix requires no storage.

We note that both the `ARKBANDPRE` and `ARKBBDPRE` preconditioner modules are hard-coded to use the SUNDIALS-supplied band `SUNMatrix` type, so the most useful information above for user-supplied `SUNMatrix` implementations is the column relating to `ARKLS` requirements.

## Chapter 10

# Linear Algebraic Solvers

For problems that require the solution of linear systems of equations, the SUNDIALS packages operate using generic linear solver modules defined through the [SUNLinearSolver](#), or “SUNLinSol”, API. This allows SUNDIALS packages to utilize any valid SUNLinSol implementation that provides a set of required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group consists of “set” routines to supply the linear solver object with functions provided by the SUNDIALS package, or for modification of solver parameters. The last group consists of “get” routines for retrieving artifacts (statistics, residual vectors, etc.) from the linear solver. All of these functions are defined in the header file `sundials/sundials_linearsolver.h`.

The implementations provided with SUNDIALS work in coordination with the SUNDIALS [N\\_Vector](#), and optionally [SUNMatrix](#), modules to provide a set of compatible data structures and solvers for the solution of linear systems using direct or iterative (matrix-based or matrix-free) methods. Moreover, advanced users can provide a customized `SUNLinearSolver` implementation to any SUNDIALS package, particularly in cases where they provide their own `N_Vector` and/or `SUNMatrix` modules.

Historically, the SUNDIALS packages have been designed to specifically leverage the use of either *direct linear solvers* or matrix-free, *scaled, preconditioned, iterative linear solvers*. However, matrix-based iterative linear solvers are also supported.

The iterative linear solvers packaged with SUNDIALS leverage scaling and preconditioning, as applicable, to balance error between solution components and to accelerate convergence of the linear solver. To this end, instead of solving the linear system  $Ax = b$  directly, these apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{10.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{10.2}$$

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  $P_2x$ .

SUNDIALS solvers request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance, i.e.,

$$\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \text{tol}.$$

When provided an iterative SUNLinSol implementation that does not support the scaling matrices  $S_1$  and  $S_2$ , the SUNDIALS packages will adjust the value of tol accordingly (see the iterative linear tolerance section that follows for more details). In this case, they instead request that iterative linear solvers stop based on the criterion

$$\|P_1^{-1}b - P_1^{-1}Ax\|_2 < \text{tol}.$$

We note that the corresponding adjustments to tol in this case may not be optimal, in that they cannot balance error between specific entries of the solution  $x$ , only the aggregate error in the overall solution vector.

We further note that not all of the SUNDIALS-provided iterative linear solvers support the full range of the above options (e.g., separate left/right preconditioning), and that some of the SUNDIALS packages only utilize a subset of these options. Further details on these exceptions are described in the documentation for each SUNLinearSolver implementation, or for each SUNDIALS package.

For users interested in providing their own SUNLinSol module, the following section presents the SUNLinSol API and its implementation beginning with the definition of SUNLinSol functions in §10.1.1 – §10.1.3. This is followed by the definition of functions supplied to a linear solver implementation in §10.1.4. The linear solver return codes are described in Table 10.1. The SUNLinearSolver type and the generic SUNLinSol module are defined in §10.1.6. §10.1.8 lists the requirements for supplying a custom SUNLinSol module and discusses some intended use cases. Users wishing to supply their own SUNLinSol module are encouraged to use the SUNLinSol implementations provided with SUNDIALS as a template for supplying custom linear solver modules. The section that then follows describes the SUNLinSol functions required by this SUNDIALS package, and provides additional package specific details. Then the remaining sections of this chapter present the SUNLinSol modules provided with SUNDIALS.

## 10.1 The SUNLinearSolver API

The SUNLinSol API defines several linear solver operations that enable SUNDIALS packages to utilize this API. These functions can be divided into three categories. The first are the core linear solver functions. The second consist of “set” routines to supply the linear solver with functions provided by the SUNDIALS packages and to modify solver parameters. The final group consists of “get” routines for retrieving linear solver statistics. All of these functions are defined in the header file `sundials/sundials_linearsolver.h`.

### 10.1.1 SUNLinearSolver core functions

The core linear solver functions consist of two **required** functions: `SUNLinSolGetType()` returns the linear solver type, and `SUNLinSolSolve()` solves the linear system  $Ax = b$ .

The remaining **optional** functions return the solver ID (`SUNLinSolGetID()`), initialize the linear solver object once all solver-specific options have been set (`SUNLinSolInitialize()`), set up the linear solver object to utilize an updated matrix  $A$  (`SUNLinSolSetup()`), and destroy a linear solver object (`SUNLinSolFree()`).

enum **SUNLinearSolver\_Type**

An identifier indicating the type of linear solver.

#### Note

See §10.1.8.1 for more information on intended use cases corresponding to the linear solver type.

**enumerator `SUNLINEARSOLVER_DIRECT`**

The linear solver requires a matrix, and computes an “exact” solution to the linear system defined by that matrix.

**enumerator `SUNLINEARSOLVER_ITERATIVE`**

The linear solver does not require a matrix (though one may be provided), and computes an inexact solution to the linear system using a matrix-free iterative algorithm. That is it solves the linear system defined by the package-supplied `ATimes` routine (see `SUNLinSolSetATimes()` below), even if that linear system differs from the one encoded in the matrix object (if one is provided). As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.

**enumerator `SUNLINEARSOLVER_MATRIX_ITERATIVE`**

The linear solver module requires a matrix, and computes an inexact solution to the linear system defined by that matrix using an iterative algorithm. That is it solves the linear system defined by the matrix object even if that linear system differs from that encoded by the package-supplied `ATimes` routine. As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.

**enumerator `SUNLINEARSOLVER_MATRIX_EMBEDDED`**

The linear solver sets up and solves the specified linear system at each linear solve call. Any matrix-related data structures are held internally to the linear solver itself, and are not provided by the SUNDIALS package.

***SUNLinearSolver\_Type* `SUNLinSolGetType(SUNLinearSolver LS)`**

Returns the *SUNLinearSolver\_Type* type identifier for the linear solver.

**Usage:**

```
type = SUNLinSolGetType(LS);
```

***SUNLinearSolver\_ID* `SUNLinSolGetID(SUNLinearSolver LS)`**

Returns a non-negative linear solver identifier (of type `int`) for the linear solver *LS*.

**Return value:**

Non-negative linear solver identifier (of type `int`), defined by the enumeration `SUNLinearSolver_ID`, with values shown in Table 10.2 and defined in the `sundials_linearsolver.h` header file.

**Usage:**

```
id = SUNLinSolGetID(LS);
```

**Note**

It is recommended that a user-supplied `SUNLinearSolver` return the `SUNLINEARSOLVER_CUSTOM` identifier.

***SUNErrCode* `SUNLinSolInitialize(SUNLinearSolver LS)`**

Performs linear solver initialization (assuming that all solver-specific options have been set).

**Return value:**

A *SUNErrCode*.

**Usage:**

```
retval = SUNLinSolInitialize(LS);
```

int **SUNLinSolSetup**(*SUNLinearSolver* LS, *SUNMatrix* A)

Performs any linear solver setup needed, based on an updated system *SUNMatrix* A. This may be called frequently (e.g., with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves.

**Return value:**

Zero for a successful call, a positive value for a recoverable failure, and a negative value for an unrecoverable failure. Ideally this should return one of the generic error codes listed in [Table 10.1](#).

**Usage:**

```
retval = SUNLinSolSetup(LS, A);
```

int **SUNLinSolSolve**(*SUNLinearSolver* LS, *SUNMatrix* A, *N\_Vector* x, *N\_Vector* b, *sunrealtype* tol)

This *required* function solves a linear system  $Ax = b$ .

**Arguments:**

- *LS* – a *SUNLinSol* object.
- *A* – a *SUNMatrix* object.
- *x* – an *N\_Vector* object containing the initial guess for the solution of the linear system on input, and the solution to the linear system upon return.
- *b* – an *N\_Vector* object containing the linear system right-hand side.
- *tol* – the desired linear solver tolerance.

**Return value:**

Zero for a successful call, a positive value for a recoverable failure, and a negative value for an unrecoverable failure. Ideally this should return one of the generic error codes listed in [Table 10.1](#).

**Notes:**

**Direct solvers:** can ignore the *tol* argument.

**Matrix-free solvers:** (those that identify as *SUNLINEARSOLVER\_ITERATIVE*) can ignore the *SUNMatrix* input *A*, and should rely on the matrix-vector product function supplied through the routine *SUNLinSolSetATimes()*.

**Iterative solvers:** (those that identify as *SUNLINEARSOLVER\_ITERATIVE* or *SUNLINEARSOLVER\_MATRIX\_ITERATIVE*) should attempt to solve to the specified tolerance *tol* in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

**Matrix-embedded solvers:** should ignore the *SUNMatrix* input *A* as this will be NULL. It is assumed that within this function, the solver will call interface routines from the relevant SUNDIALS package to directly form the linear system matrix *A*, and then solve  $Ax = b$  before returning with the solution *x*.

**Usage:**

```
retval = SUNLinSolSolve(LS, A, x, b, tol);
```

*SUNErrCode* **SUNLinSolFree**(*SUNLinearSolver* LS)

Frees memory allocated by the linear solver.

**Return value:**

A *SUNErrCode*.

**Usage:**



```
retval = SUNLinSolFree(LS);
```

### 10.1.2 SUNLinearSolver “set” functions

The following functions supply linear solver modules with functions defined by the SUNDIALS packages and modify solver parameters. Only the routine for setting the matrix-vector product routine is required, and even then is only required for matrix-free linear solver modules. Otherwise, all other set functions are optional. SUNLinSol implementations that do not provide the functionality for any optional routine should leave the corresponding function pointer NULL instead of supplying a dummy routine.

*SUNErrCode* **SUNLinSolSetOptions**(*SUNLinearSolver* S, const char \*LSid, const char \*file\_name, int argc, char \*argv[])

This *optional* routine sets SUNLinearSolver options from an array of strings or a file.

#### Parameters

- **S** – the *SUNLinearSolver* object.
- **LSid** – the prefix for options to read. The default is “sunlinearsolver”.
- **file\_name** – the name of a file containing options to read. If this is NULL or an empty string, “”, then no file is read.
- **argc** – length of the argv array.
- **argv** – an array of strings containing the options to set and their values.

#### Returns

*SUNErrCode* indicating success or failure.

#### Note

The argc and argv arguments are typically those supplied to the user’s main routine however, this is not required. The inputs are left unchanged by *SUNLinSolSetOptions()*.

If the LSid argument is NULL, then the default prefix, sunlinearsolver, must be used for all SUNLinearSolver options. Whether LSid is supplied or not, a “.” must be used to separate an option key from the prefix. For example, when using the default LSid, the option sunlinearsolver.zero\_guess can be used to inform an iterative linear solver to use a zero-valued initial guess. When using a combination of SUNLinearSolver objects (e.g., for system and mass matrices within ARKStep), it is recommended that users call *SUNLinSolSetOptions()* for each linear solver using distinct LSid inputs, so that each solver object can be configured separately.

SUNLinearSolver options set via command-line arguments to *SUNLinSolSetOptions()* will overwrite any previously-set values. Options are set in the order they are given in argv and, if an option with the same prefix appears multiple times in argv, the value of the last occurrence will be used.

The supported options are documented within each SUNLinearSolver “set” routine. For options that take a *sunbooleantype* as input, use 1 to indicate true and 0 for false.

#### Warning

This function is not available in the Fortran interface.

File-based options are not yet supported, so the `file_name` argument should be set to either `NULL` or the empty string `""`.

Added in version 7.5.0.

*SUNErrCode* **SUNLinSolSetATimes**(*SUNLinearSolver* LS, void \*A\_data, *SUNATimesFn* ATimes)

*Required for matrix-free linear solvers* (otherwise optional).

Provides a *SUNATimesFn* function pointer, as well as a `void*` pointer to a data structure used by this routine, to the linear solver object *LS*. SUNDIALS packages 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.

**Return value:**

A *SUNErrCode*.

**Usage:**

```
retval = SUNLinSolSetATimes(LS, A_data, ATimes);
```

*SUNErrCode* **SUNLinSolSetPreconditioner**(*SUNLinearSolver* LS, void \*P\_data, *SUNPSetupFn* Pset, *SUNPSolveFn* Psol)

This *optional* routine provides *SUNPSetupFn* and *SUNPSolveFn* function pointers that implement the preconditioner solves  $P_1^{-1}$  and  $P_2^{-1}$  from (10.2). This routine is called by a SUNDIALS package, which provides translation between the generic *Pset* and *Psol* calls and the package- or user-supplied routines.

**Return value:**

A *SUNErrCode*.

**Usage:**

```
retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);
```

*SUNErrCode* **SUNLinSolSetScalingVectors**(*SUNLinearSolver* LS, *N\_Vector* s1, *N\_Vector* s2)

This *optional* routine provides left/right scaling vectors for the linear system solve. Here, *s1* and *s2* are vectors of positive scale factors containing the diagonal of the matrices  $S_1$  and  $S_2$  from (10.2), respectively. Neither vector needs to be tested for positivity, and a `NULL` argument for either indicates that the corresponding scaling matrix is the identity.

**Return value:**

A *SUNErrCode*.

**Usage:**

```
retval = SUNLinSolSetScalingVectors(LS, s1, s2);
```

**Warning**

The vectors *s1* and *s2* should not be modified.

*SUNErrCode* **SUNLinSolSetZeroGuess**(*SUNLinearSolver* LS, *sunbooleantype* onoff)

This *optional* routine indicates if the upcoming *SUNLinSolSolve()* call will be made with a zero initial guess (`SUNTRUE`) or a non-zero initial guess (`SUNFALSE`).

**Return value:**

A *SUNErrCode*.

**Usage:**

```
retval = SUNLinSolSetZeroGuess(LS, onoff);
```

**Notes:**

It is assumed that the initial guess status is not retained across calls to *SUNLinSolSolve()*. As such, the linear solver interfaces in each of the SUNDIALS packages call *SUNLinSolSetZeroGuess()* prior to each call to *SUNLinSolSolve()*.

If supported by the SUNLinearSolver implementation, this routine will be called by *SUNLinSolSetOptions()* when using the key “LSid.zero\_guess”.

### 10.1.3 SUNLinearSolver “get” functions

The following functions allow SUNDIALS packages to retrieve results from a linear solve. *All routines are optional.*

int **SUNLinSolNumIters**(*SUNLinearSolver* LS)

This *optional* routine should return the number of linear iterations performed in the most-recent “solve” call.

**Usage:**

```
its = SUNLinSolNumIters(LS);
```

*sunrealtype* **SUNLinSolResNorm**(*SUNLinearSolver* LS)

This *optional* routine should return the final residual norm from the most-recent “solve” call.

**Usage:**

```
rnorm = SUNLinSolResNorm(LS);
```

*N\_Vector* **SUNLinSolResid**(*SUNLinearSolver* LS)

If an iterative method computes the preconditioned initial residual and returns with a successful solve without performing any iterations (i.e., either the initial guess or the preconditioner is sufficiently accurate), then this *optional* routine may be called by the SUNDIALS package. This routine should return the *N\_Vector* containing the preconditioned initial residual vector.

**Usage:**

```
rvec = SUNLinSolResid(LS);
```

**Notes:**

Since *N\_Vector* is actually a pointer, and the results are not modified, this routine should *not* require additional memory allocation. If the *SUNLinSol* object does not retain a vector for this purpose, then this function pointer should be set to NULL in the implementation.

*sunindextype* **SUNLinSolLastFlag**(*SUNLinearSolver* LS)

This *optional* routine should return the last error flag encountered within the linear solver. Although not called by the SUNDIALS packages directly, this may be called by the user to investigate linear solver issues after a failed solve.

**Usage:**

```
lflag = SUNLinSolLastFlag(LS);
```

*SUNErrCode* **SUNLinSolSpace**(*SUNLinearSolver* LS, long int \*lenrwLS, long int \*leniwLS)

This *optional* routine should return the storage requirements for the linear solver *LS*:

- *lrw* is a long int containing the number of sunrealtype words
- *liw* is a long int containing the number of integer words.

This function is advisory only, for use by users to help determine their total space requirements.

**Return value:**

A *SUNErrCode*.

**Usage:**

```
retval = SUNLinSolSpace(LS, &lrw, &liw);
```

Deprecated since version 7.3.0: Work space functions will be removed in version 8.0.0.

### 10.1.4 Functions provided by SUNDIALS packages

To interface with SUNLinSol modules, the SUNDIALS packages supply a variety of routines for evaluating the matrix-vector product, and setting up and applying the preconditioner. These package-provided routines translate between the user-supplied ODE, DAE, or nonlinear systems and the generic linear solver API. The function types for these routines are defined in the header file `sundials/sundials_iterative.h`, and are described below.

typedef int (\***SUNATimesFn**)(void \*A\_data, *N\_Vector* v, *N\_Vector* z)

Computes the action of a matrix on a vector, performing the operation  $z \leftarrow Av$ . Memory for *z* will already be allocated prior to calling this function. The parameter *A\_data* is a pointer to any information about *A* which the function needs in order to do its job. The vector *v* should be left unchanged.

**Return value:**

Zero for a successful call, and non-zero upon failure.

typedef int (\***SUNPSetupFn**)(void \*P\_data)

Sets up any requisite problem data in preparation for calls to the corresponding *SUNPSolveFn*.

**Return value:**

Zero for a successful call, and non-zero upon failure.

typedef int (\***SUNPSolveFn**)(void \*P\_data, *N\_Vector* r, *N\_Vector* z, *sunrealtype* tol, int lr)

Solves the preconditioner equation  $Pz = r$  for the vector *z*. Memory for *z* will already be allocated prior to calling this function. The parameter *P\_data* is a pointer to any information about *P* which the function needs in order to do its job (set up by the corresponding *SUNPSetupFn*). The parameter *lr* is input, and indicates whether *P* is to be taken as the left or right preconditioner: *lr* = 1 for left and *lr* = 2 for right. If preconditioning is on one side only, *lr* can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$\|Pz - r\|_{\text{wrms}} < \text{tol}$$

where the error weight vector for the WRMS norm may be accessed from the main package memory structure. The vector *r* should not be modified by the *SUNPSolveFn*.

**Return value:**

Zero for a successful call, a negative value for an unrecoverable failure condition, or a positive value for a recoverable failure condition (thus the calling routine may reattempt the solution after updating preconditioner data).

### 10.1.5 SUNLinearSolver return codes

The functions provided to SUNLinSol modules by each SUNDIALS package, and functions within the SUNDIALS-provided SUNLinSol implementations, utilize a common set of return codes, listed in Table 10.1. These adhere to a common pattern:

- 0 indicates success
- a positive value corresponds to a recoverable failure, and
- a negative value indicates a non-recoverable failure.

Aside from this pattern, the actual values of each error code provide additional information to the user in case of a linear solver failure.

Table 10.1: SUNLinSol error codes

Error code	Value	Meaning
SUN_SUCCESS	0	successful call or converged solve
SUNLS_ATIMES_NULL	-804	the <code>ATimes</code> function is NULL
SUNLS_ATIMES_FAIL_-UNREC	-805	an unrecoverable failure occurred in the <code>ATimes</code> routine
SUNLS_PSET_FAIL_-UNREC	-806	an unrecoverable failure occurred in the <code>Pset</code> routine
SUNLS_PSOLVE_NULL	-807	the preconditioner solve function is NULL
SUNLS_PSOLVE_FAIL_-UNREC	-808	an unrecoverable failure occurred in the <code>Psolve</code> routine
SUNLS_GS_FAIL	-810	a failure occurred during Gram-Schmidt orthogonalization (SPGMR/SPFGMR)
SUNLS_QRSOL_FAIL	-811	a singular $SR$ matrix was encountered in a QR factorization (SPGMR/SPFGMR)
SUNLS_RES_REDUCED	801	an iterative solver reduced the residual, but did not converge to the desired tolerance
SUNLS_CONV_FAIL	802	an iterative solver did not converge (and the residual was not reduced)
SUNLS_ATIMES_FAIL_-REC	803	a recoverable failure occurred in the <code>ATimes</code> routine
SUNLS_PSET_FAIL_REC	804	a recoverable failure occurred in the <code>Pset</code> routine
SUNLS_PSOLVE_FAIL_-REC	805	a recoverable failure occurred in the <code>Psolve</code> routine
SUNLS_PACKAGE_FAIL_-REC	806	a recoverable failure occurred in an external linear solver package
SUNLS_QRFACT_FAIL	807	a singular matrix was encountered during a QR factorization (SPGMR/SPFGMR)
SUNLS_LUFACT_FAIL	808	a singular matrix was encountered during a LU factorization

### 10.1.6 The generic SUNLinearSolver module

SUNDIALS packages interact with linear solver implementations through the `SUNLinearSolver` class. A `SUNLinearSolver` is a pointer to the `_generic_SUNLinearSolver` structure:

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver
```

```
struct _generic_SUNLinearSolver
```

The structure defining the SUNDIALS linear solver class.

void **\*content**

Pointer to the linear solver-specific member data

*SUNLinearSolver\_Ops* **ops**

A virtual table of linear solver operations provided by a specific implementation

*SUNContext* **sunctx**

The SUNDIALS simulation context

The virtual table structure is defined as

```
typedef struct _generic_SUNLinearSolver_Ops *SUNLinearSolver_Ops
```

```
struct _generic_SUNLinearSolver_Ops
```

The structure defining *SUNLinearSolver* operations.

*SUNLinearSolver\_Type* (**\*gettype**)(*SUNLinearSolver*)

The function implementing *SUNLinSolGetType()*

*SUNLinearSolver\_ID* (**\*getid**)(*SUNLinearSolver*)

The function implementing *SUNLinSolGetID()*

*SUNErrCode* (**\*setatimes**)(*SUNLinearSolver*, void\*, *SUNATimesFn*)

The function implementing *SUNLinSolSetATimes()*

*SUNErrCode* (**\*setpreconditioner**)(*SUNLinearSolver*, void\*, *SUNPSSetupFn*, *SUNPSolveFn*)

The function implementing *SUNLinSolSetPreconditioner()*

*SUNErrCode* (**\*setscalingvectors**)(*SUNLinearSolver*, *N\_Vector*, *N\_Vector*)

The function implementing *SUNLinSolSetScalingVectors()*

*SUNErrCode* (**\*setoptions**)(*SUNLinearSolver*, const char \*LSid, const char \*file\_name, int argc, char \*argv[])

The function implementing *SUNLinSolSetOptions()*

*SUNErrCode* (**\*setzeroguess**)(*SUNLinearSolver*, *sunbooleantype*)

The function implementing *SUNLinSolSetZeroGuess()*

*SUNErrCode* (**\*initialize**)(*SUNLinearSolver*)

The function implementing *SUNLinSolInitialize()*

int (**\*setup**)(*SUNLinearSolver*, *SUNMatrix*)

The function implementing *SUNLinSolSetup()*

int (**\*solve**)(*SUNLinearSolver*, *SUNMatrix*, *N\_Vector*, *N\_Vector*, *sunrealtype*)

The function implementing *SUNLinSolSolve()*

int (**\*numiters**)(*SUNLinearSolver*)

The function implementing *SUNLinSolNumIters()*

*sunrealtype* (**\*resnorm**)(*SUNLinearSolver*)

The function implementing *SUNLinSolResNorm()*

*sunindextype* (**\*lastflag**)(*SUNLinearSolver*)

The function implementing *SUNLinSolLastFlag()*

*SUNErrCode* (**\*space**)(*SUNLinearSolver*, long int\*, long int\*)

The function implementing *SUNLinSolSpace()*

*N\_Vector* (\***resid**)(*SUNLinearSolver*)

The function implementing *SUNLinSolResid()*

*SUNErrCode* (\***free**)(*SUNLinearSolver*)

The function implementing *SUNLinSolFree()*

The generic *SUNLinSol* class defines and implements the linear solver operations defined in §10.1.1 – §10.1.3. These routines are in fact only wrappers to the linear solver operations defined by a particular *SUNLinSol* implementation, which are accessed through the *ops* field of the *SUNLinearSolver* structure. To illustrate this point we show below the implementation of a typical linear solver operation from the *SUNLinearSolver* base class, namely *SUNLinSolInitialize()*, that initializes a *SUNLinearSolver* object for use after it has been created and configured, and returns a flag denoting a successful or failed operation:

```
int SUNLinSolInitialize(SUNLinearSolver S)
{
    return ((int) S->ops->initialize(S));
}
```

### 10.1.7 Compatibility of *SUNLinearSolver* modules

Not all *SUNLinearSolver* implementations are compatible with all *SUNMatrix* and *N\_Vector* implementations provided in SUNDIALS. More specifically, all of the SUNDIALS iterative linear solvers (*SPGMR*, *SPFGMR*, *SPBCGS*, *SPTFQMR*, and *PCG*) are compatible with all of the SUNDIALS *N\_Vector* modules, but the matrix-based direct *SUNLinSol* modules are specifically designed to work with distinct *SUNMatrix* and *N\_Vector* modules. In the list below, we summarize the compatibility of each matrix-based *SUNLinearSolver* module with the various *SUNMatrix* and *N\_Vector* modules. For a more thorough discussion of these compatibilities, we defer to the documentation for each individual *SUNLinSol* module in the sections that follow.

- *Dense*
  - *SUNMatrix*: *Dense* or user-supplied
  - *N\_Vector*: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *LapackDense*
  - *SUNMatrix*: *Dense* or user-supplied
  - *N\_Vector*: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *Band*
  - *SUNMatrix*: *Band* or user-supplied
  - *N\_Vector*: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *LapackBand*
  - *SUNMatrix*: *Band* or user-supplied
  - *N\_Vector*: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *KLU*
  - *SUNMatrix*: *Sparse* or user-supplied
  - *N\_Vector*: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *SuperLU\_MT*
  - *SUNMatrix*: *Sparse* or user-supplied

- N\_Vector: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *SuperLU\_Dist*
  - SUNMatrix: *SLUNRLOC* or user-supplied
  - N\_Vector: *Serial*, *OpenMP*, *Pthreads*, *Parallel*, *\*hypr\**, *PETSc*, or user-supplied
- *Magma Dense*
  - SUNMatrix: *Magma Dense* or user-supplied
  - N\_Vector: *HIP*, *RAJA*, or user-supplied
- *OneMKL Dense*
  - SUNMatrix: *One MKL Dense* or user-supplied
  - N\_Vector: *SYCL*, *RAJA*, or user-supplied
- *cuSolverSp batchQR*
  - SUNMatrix: *cuSparse* or user-supplied
  - N\_Vector: *CUDA*, *RAJA*, or user-supplied

### 10.1.8 Implementing a custom SUNLinearSolver module

A particular implementation of the SUNLinearSolver module must:

- Specify the *content* field of the SUNLinSol module.
- Define and implement the required linear solver operations.

Note
The names of these routines should be unique to that implementation in order to permit using more than one SUNLinSol module (each with different SUNLinearSolver internal data representations) in the same code.

- Define and implement user-callable constructor and destructor routines to create and free a SUNLinearSolver with the new *content* field and with *ops* pointing to the new linear solver operations.

We note that the function pointers for all unsupported optional routines should be set to NULL in the *ops* structure. This allows the SUNDIALS package that is using the SUNLinSol object to know whether the associated functionality is supported.

To aid in the creation of custom SUNLinearSolver modules the generic SUNLinearSolver module provides the utility function `SUNLinSolNewEmpty()`. When used in custom SUNLinearSolver constructors this function will ease the introduction of any new optional linear solver operations to the SUNLinearSolver API by ensuring that only required operations need to be set.

*SUNLinearSolver* **SUNLinSolNewEmpty**(*SUNContext* sunctx)

This function allocates a new generic SUNLinearSolver object and initializes its content pointer and the function pointers in the operations structure to NULL.

**Return value:**

If successful, this function returns a SUNLinearSolver object. If an error occurs when allocating the object, then this routine will return NULL.



void **SUNLinSolFreeEmpty**(*SUNLinearSolver* LS)

This routine frees the generic `SUNLinearSolver` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will free it as well.

**Arguments:**

- *LS* – a `SUNLinearSolver` object

Additionally, a `SUNLinearSolver` implementation *may* do the following:

- Define and implement additional user-callable “set” routines acting on the `SUNLinearSolver`, e.g., for setting various configuration options to tune the linear solver for a particular problem.
- Provide additional user-callable “get” routines acting on the `SUNLinearSolver` object, e.g., for returning various solve statistics.

enum **SUNLinearSolver\_ID**

Each `SUNLinSol` implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in [Table 10.2](#). It is recommended that a user-supplied `SUNLinSol` implementation use the `SUNLINEAR-SOLVER_CUSTOM` identifier.

Table 10.2: Identifiers associated with `SUNLinearSolver` modules supplied with SUNDIALS

SUNLinSol ID	Linear solver type	ID Value
SUNLINEARSOLVER_BAND	Banded direct linear solver (internal)	0
SUNLINEARSOLVER_DENSE	Dense direct linear solver (internal)	1
SUNLINEARSOLVER_KLU	Sparse direct linear solver (KLU)	2
SUNLINEARSOLVER_LAPACKBAND	Banded direct linear solver (LAPACK)	3
SUNLINEARSOLVER_LAPACKDENSE	Dense direct linear solver (LAPACK)	4
SUNLINEARSOLVER_PCG	Preconditioned conjugate gradient iterative solver	5
SUNLINEARSOLVER_SPBCGS	Scaled-preconditioned BiCGStab iterative solver	6
SUNLINEARSOLVER_SPGMR	Scaled-preconditioned FGMRES iterative solver	7
SUNLINEARSOLVER_SPGMR	Scaled-preconditioned GMRES iterative solver	8
SUNLINEARSOLVER_SPTFQMR	Scaled-preconditioned TFQMR iterative solver	9
SUNLINEARSOLVER_SUPERLUDIST	Parallel sparse direct linear solver (SuperLU_Dist)	10
SUNLINEARSOLVER_SUPERLUMT	Threaded sparse direct linear solver (SuperLU_MT)	11
SUNLINEARSOLVER_CUSOLVERSP_BATCHQR	Sparse direct linear solver (CUDA)	12
SUNLINEARSOLVER_MAGMADENSE	Dense or block-dense direct linear solver (MAGMA)	13
SUNLINEARSOLVER_ONEMKLDENSE	Dense or block-dense direct linear solver (OneMKL)	14
SUNLINEARSOLVER_CUSTOM	User-provided custom linear solver	15

### 10.1.8.1 Intended use cases

The `SUNLinSol` and `SUNMATRIX` APIs are designed to require a minimal set of routines to ease interfacing with custom or third-party linear solver libraries. Many external solvers provide routines with similar functionality and thus may require minimal effort to wrap within custom `SUNMATRIX` and `SUNLinSol` implementations. As SUNDIALS

packages utilize generic SUNLinSol modules they may naturally leverage user-supplied SUNLinearSolver implementations, thus there exist a wide range of possible linear solver combinations. Some intended use cases for both the SUNDIALS-provided and user-supplied SUNLinSol modules are discussed in the sections below.

### Direct linear solvers

Direct linear solver modules require a matrix and compute an “exact” solution to the linear system *defined by the matrix*. SUNDIALS packages strive to amortize the high cost of matrix construction by reusing matrix information for multiple nonlinear iterations or time steps. As a result, each package’s linear solver interface recomputes matrix information as infrequently as possible.

Alternative matrix storage formats and compatible linear solvers that are not currently provided by, or interfaced with, SUNDIALS can leverage this infrastructure with minimal effort. To do so, a user must implement custom SUNMATRIX and SUNLinSol wrappers for the desired matrix format and/or linear solver following the APIs described in §9 and §10. *This user-supplied SUNLinSol module must then self-identify as having SUNLINEARSOLVER\_DIRECT type.*

### Matrix-free iterative linear solvers

Matrix-free iterative linear solver modules do not require a matrix, and instead compute an inexact solution to the linear system *defined by the package-supplied ATimes routine*. SUNDIALS supplies multiple scaled, preconditioned iterative SUNLinSol modules that support scaling, allowing packages to handle non-dimensionalization, and users to define variables and equations as natural in their applications. However, for linear solvers that do not support left/right scaling, SUNDIALS packages must instead adjust the tolerance supplied to the linear solver to compensate (see the iterative linear tolerance section that follows for more details) – this strategy may be non-optimal since it cannot handle situations where the magnitudes of different solution components or equations vary dramatically within a single application.

To utilize alternative linear solvers that are not currently provided by, or interfaced with, SUNDIALS a user must implement a custom SUNLinSol wrapper for the linear solver following the API described in §10. *This user-supplied SUNLinSol module must then self-identify as having SUNLINEARSOLVER\_ITERATIVE type.*

### Matrix-based iterative linear solvers (reusing $A$ )

Matrix-based iterative linear solver modules require a matrix and compute an inexact solution to the linear system *defined by the matrix*. This matrix will be updated infrequently and reused across multiple solves to amortize the cost of matrix construction. As in the direct linear solver case, only thin SUNMATRIX and SUNLinSol wrappers for the underlying matrix and linear solver structures need to be created to utilize such a linear solver. *This user-supplied SUNLinSol module must then self-identify as having SUNLINEARSOLVER\_MATRIX\_ITERATIVE type.*

At present, SUNDIALS has one example problem that uses this approach for wrapping a structured-grid matrix, linear solver, and preconditioner from the *hypre* library; this may be used as a template for other customized implementations (see `examples/arkode/CXX_parhyp/ark_heat2D_hypre.cpp`).

### Matrix-based iterative linear solvers (current $A$ )

For users who wish to utilize a matrix-based iterative linear solver where the matrix is *purely for preconditioning* and the linear system is *defined by the package-supplied ATimes routine*, we envision two current possibilities.

The preferred approach is for users to employ one of the SUNDIALS scaled, preconditioned iterative linear solver implementations (`SUNLinSol_SPGMR()`, `SUNLinSol_SPFGMR()`, `SUNLinSol_SPBCGS()`, `SUNLinSol_SPTFQMR()`, or `SUNLinSol_PCG()`) as the outer solver. The creation and storage of the preconditioner matrix, and interfacing with the corresponding matrix-based linear solver, can be handled through a package’s preconditioner “setup” and “solve”

functionality without creating SUNMATRIX and SUNLinSol implementations. This usage mode is recommended primarily because the SUNDIALS-provided modules support variable and equation scaling as described above.

A second approach supported by the linear solver APIs is as follows. If the SUNLinSol implementation is matrix-based, *self-identifies as having* `SUNLINEARSOLVER_ITERATIVE` type, and *also provides a non-NULL* `SUNLinSolSetAtimes()` routine, then each SUNDIALS package will call that routine to attach its package-specific matrix-vector product routine to the SUNLinSol object. The SUNDIALS package will then call the SUNLinSol-provided `SUNLinSolSetup()` routine (infrequently) to update matrix information, but will provide current matrix-vector products to the SUNLinSol implementation through the package-supplied `SUNAtimesFn` routine.

### Application-specific linear solvers with embedded matrix structure

Many applications can exploit additional linear system structure arising from the implicit couplings in their model equations. In certain circumstances, the linear solve  $Ax = b$  may be performed without the need for a global system matrix  $A$ , as the unfurmed  $A$  may be block diagonal or block triangular, and thus the overall linear solve may be performed through a sequence of smaller linear solves. In other circumstances, a linear system solve may be accomplished via specialized fast solvers, such as the fast Fourier transform, fast multipole method, or treecode, in which case no matrix structure may be explicitly necessary. In many of the above situations, construction and preprocessing of the linear system matrix  $A$  may be inexpensive, and thus increased performance may be possible if the current linear system information is used within every solve (instead of being lagged, as occurs with matrix-based solvers that reuse  $A$ ).

To support such application-specific situations, SUNDIALS supports user-provided linear solvers with the `SUNLINEARSOLVER_MATRIX_EMBEDDED` type. For an application to leverage this support, it should define a custom SUNLinSol implementation having this type, that only needs to implement the required `SUNLinSolGetType()` and `SUNLinSolSolve()` operations. Within `SUNLinSolSolve()`, the linear solver implementation should call package-specific interface routines (e.g., `ARKStepGetNonlinearSystemData`, `CVodeGetNonlinearSystemData`, `IDAGetNonlinearSystemData`, `ARKStepGetCurrentGamma`, `CVodeGetCurrentGamma`, `IDAGetCurrentCj`, or `MRIStepGetCurrentGamma`) to construct the relevant system matrix  $A$  (or portions thereof), solve the linear system  $Ax = b$ , and return the solution vector  $x$ .

We note that when attaching this custom SUNLinearSolver object with the relevant SUNDIALS package `SetLinearSolver` routine, the input `SUNMatrix`  $A$  should be set to `NULL`.

For templates of such user-provided “matrix-embedded” SUNLinSol implementations, see the SUNDIALS examples `ark_analytic_mels.c`, `cvAnalytic_mels.c`, `cvsAnalytic_mels.c`, `idaAnalytic_mels.c`, and `idasAnalytic_mels.c`.

## 10.2 ARKODE SUNLinearSolver interface

In Table 10.3, we list the SUNLinSol module functions used within the ARKLS interface. As with the SUNMATRIX module, we emphasize that the ARKODE user does not need to know detailed usage of linear solver functions by the ARKODE code modules in order to use ARKODE. The information is presented as an implementation detail for the interested reader.

Table 10.3: List of SUNLinSol functions called by the ARKODE linear solver interface, depending on the self-identified “type” reported from SUNLinSolGetType(). Functions marked with “X” are required; functions marked with “O” are only called if they are non-NULL in the SUNLinearSolver implementation that is being used.

Routine	DI- RECT	ITERA- TIVE	MATRIX TIVE	ITERA- TIVE	MATRIX DED	EMBED-
<code>SUNLinSolGetType()</code>	X	X	X		X	
<code>SUNLinSolSetATimes()</code>	O	X	O			
<code>SUNLinSolSetPreconditioner()</code>	O	O	O			
<code>SUNLinSolSetScalingVectors()</code>	O	O	O			
<code>SUNLinSolInitialize()</code>	X	X	X			
<code>SUNLinSolSetup()</code>	X	X	X			
<code>SUNLinSolSolve()</code>	X	X	X		X	
<code>SUNLinSolNumIters()</code> <sup>1</sup>		O	O			
<code>SUNLinSolResNorm()</code> <sup>2</sup>		O	O			
<code>SUNLinSolLastFlag()</code> <sup>3</sup>						
<code>SUNLinSolFree()</code> <sup>4</sup>						
<code>SUNLinSolSpace()</code>	O	O	O		O	

Notes:

1. `SUNLinSolNumIters()` is only used to accumulate overall iterative linear solver statistics. If it is not implemented by the SUNLinearSolver module, then ARKLS will consider all solves as requiring zero iterations.
2. Although `SUNLinSolResNorm()` is optional, if it is not implemented by the SUNLinearSolver then ARKLS will consider all solves a being *exact*.
3. Although ARKLS does not call `SUNLinSolLastFlag()` directly, this routine is available for users to query linear solver failure modes.
4. Although ARKLS does not call `SUNLinSolFree()` directly, this routine should be available for users to call when cleaning up from a simulation.

Since there are a wide range of potential SUNLinSol use cases, the following subsections describe some details of the ARKLS interface, in the case that interested users wish to develop custom SUNLinSol modules.

### 10.2.1 Lagged matrix information

If the SUNLinSol module identifies as having type `SUNLINEARSOLVER_DIRECT` or `SUNLINEARSOLVER_MATRIX_ITERATIVE`, then it solves a linear system *defined* by a `SUNMATRIX` object. ARKLS will update the matrix information infrequently according to the strategies outlined in §2.15.2.3. To this end, we differentiate between the *desired* linear system  $\mathcal{A}x = b$  with  $\mathcal{A} = (M - \gamma J)$  and the *actual* linear system

$$\tilde{\mathcal{A}}\tilde{x} = b \quad \Leftrightarrow \quad (M - \tilde{\gamma}J)\tilde{x} = b.$$

Since ARKLS updates the `SUNMATRIX` object infrequently, it is likely that  $\gamma \neq \tilde{\gamma}$ , and in turn  $\mathcal{A} \neq \tilde{\mathcal{A}}$ . Therefore, after calling the SUNLinSol-provided `SUNLinSolSolve()` routine, we test whether  $\gamma/\tilde{\gamma} \neq 1$ , and if this is the case we scale the solution  $\tilde{x}$  to obtain the desired linear system solution  $x$  via

$$x = \frac{2}{1 + \gamma/\tilde{\gamma}}\tilde{x}. \quad (10.3)$$

The motivation for this selection of the scaling factor  $c = 2/(1 + \gamma/\tilde{\gamma})$  follows the derivation in [22, 60]. In short, if we consider a stationary iteration for the linear system as consisting of a solve with  $\tilde{A}$  followed with a scaling by  $c$ , then for a linear constant-coefficient problem, the error in the solution vector will be reduced at each iteration by the error matrix  $E = I - c\tilde{A}^{-1}A$ , with a convergence rate given by the spectral radius of  $E$ . Assuming that stiff systems have a spectrum spread widely over the left half-plane,  $c$  is chosen to minimize the magnitude of the eigenvalues of  $E$ .

## 10.2.2 Iterative linear solver tolerance

If the SUNLinSol object self-identifies as having type `SUNLINEARSOLVER_ITERATIVE` or `SUNLINEARSOLVER_MATRIX_ITERATIVE`, then ARKLS will set the input tolerance `delta` as described in §2.15.3.2. However, if the iterative linear solver does not support scaling matrices (i.e., the `SUNLinSolSetScalingVectors()` routine is `NULL`), then ARKLS will attempt to adjust the linear solver tolerance to account for this lack of functionality. To this end, the following assumptions are made:

- All solution components have similar magnitude; hence the residual weight vector  $w$  used in the WRMS norm (see §2.10), corresponding to the left scaling matrix  $S_1$ , should satisfy the assumption

$$w_i \approx w_{mean}, \quad \text{for } i = 0, \dots, n-1.$$

- The SUNLinSol object uses a standard 2-norm to measure convergence.

Under these assumptions, ARKLS adjusts the linear solver convergence requirement as follows (using the notation from (10.2)):

$$\begin{aligned} & \|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \text{tol} \\ \Leftrightarrow & \|S_1 P_1^{-1} b - S_1 P_1^{-1} A x\|_2 < \text{tol} \\ \Leftrightarrow & \sum_{i=0}^{n-1} [w_i (P_1^{-1}(b - Ax))_i]^2 < \text{tol}^2 \\ \Leftrightarrow & w_{mean}^2 \sum_{i=0}^{n-1} [(P_1^{-1}(b - Ax))_i]^2 < \text{tol}^2 \\ \Leftrightarrow & \sum_{i=0}^{n-1} [(P_1^{-1}(b - Ax))_i]^2 < \left(\frac{\text{tol}}{w_{mean}}\right)^2 \\ \Leftrightarrow & \|P_1^{-1}(b - Ax)\|_2 < \frac{\text{tol}}{w_{mean}} \end{aligned}$$

Therefore we compute the tolerance scaling factor

$$w_{mean} = \|w\|_2 / \sqrt{n}$$

and supply the scaled tolerance `delta = tol/wmean` to the SUNLinSol object.

## 10.2.3 Providing a custom SUNLinearSolver

In certain instances, users may wish to provide a custom SUNLinSol implementation to ARKODE in order to leverage the structure of a problem. While the “standard” API for these routines is typically sufficient for most users, others may need additional ARKODE-specific information on top of what is provided. For these purposes, we note the following advanced output functions available in ARKStep and MRISStep:

**ARKStep advanced outputs:** when solving the Newton nonlinear system of equations in predictor-corrector form,

$$\begin{aligned} G(z_{cor}) &\equiv z_{cor} - \gamma f^I(t_{n,i}^I, z_i) - \tilde{a}_i = 0 & [M = I], \\ G(z_{cor}) &\equiv M z_{cor} - \gamma f^I(t_{n,i}^I, z_i) - \tilde{a}_i = 0 & [M \text{ static}], \\ G(z_{cor}) &\equiv M(t_{n,i}^I)(z_{cor} - \tilde{a}_i) - \gamma f^I(t_{n,i}^I, z_i) = 0 & [M \text{ time-dependent}]. \end{aligned}$$

- [`ARKStepGetCurrentTime\(\)`](#) – when called within the computation of a step (i.e., within a solve) this returns  $t_{n,i}^I$ . Otherwise the current internal solution time is returned.
- [`ARKStepGetCurrentState\(\)`](#) – when called within the computation of a step (i.e., within a solve) this returns the current stage vector  $z_i = z_{cor} + z_{pred}$ . Otherwise the current internal solution is returned.
- [`ARKStepGetCurrentGamma\(\)`](#) – returns  $\gamma$ .
- [`ARKStepGetCurrentMassMatrix\(\)`](#) – returns  $M(t)$ .
- [`ARKStepGetNonlinearSystemData\(\)`](#) – returns  $z_i$ ,  $z_{pred}$ ,  $f^I(t_{n,i}^I, y_{cur})$ ,  $\tilde{a}_i$ , and  $\gamma$ .

**MRISStep advanced outputs:** when solving the Newton nonlinear system of equations in predictor-corrector form,

$$G(z_{cor}) \equiv z_{cor} - \gamma f^I(t_{n,i}^S, z_i) - \tilde{a}_i = 0$$

- [`MRISStepGetCurrentTime\(\)`](#) – when called within the computation of a step (i.e., within a solve) this returns  $t_{n,i}^S$ . Otherwise the current internal solution time is returned.
- [`MRISStepGetCurrentState\(\)`](#) – when called within the computation of a step (i.e., within a solve) this returns the current stage vector  $z_i = z_{cor} + z_{pred}$ . Otherwise the current internal solution is returned.
- [`MRISStepGetCurrentGamma\(\)`](#) – returns  $\gamma$ .
- [`MRISStepGetNonlinearSystemData\(\)`](#) – returns  $z_i$ ,  $z_{pred}$ ,  $f^I(t_{n,i}^I, y_{cur})$ ,  $\tilde{a}_i$ , and  $\gamma$ .

## 10.3 The SUNLinSol\_Band Module

The SUNLinSol\_Band implementation of the SUNLinearSolver class 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).

### 10.3.1 SUNLinSol\_Band Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_band.h`. The SUNLinSol\_Band module is accessible from all SUNDIALS packages *without* linking to the `libsundials_sunlinsolband` module library.

The SUNLinSol\_Band module provides the following user-callable constructor routine:

[`SUNLinearSolver`](#) **SUNLinSol\_Band**(*N\_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)

This function creates and allocates memory for a band SUNLinearSolver.

**Arguments:**

- y – vector used to determine the linear system size
- A – matrix used to assess compatibility
- sunctx – the [`SUNContext`](#) object (see §4.2)

**Return value:**

New SUNLinSol\_Band object, or NULL if either A or y are incompatible.

**Notes:**

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.

### 10.3.2 SUNLinSol\_Band Description

The SUNLinSol\_Band module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

- `N` - size of the linear system,
- `pivots` - index array for partial pivoting in  $LU$  factorization,
- `last_flag` - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The “setup” call performs an  $LU$  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 `m1`, then the upper triangular factor  $U$  can have upper bandwidth as big as `smu = MIN(N-1, mu+m1)`. The lower triangular factor  $L$  has lower bandwidth `m1`.

The SUNLinSol\_Band module defines band implementations of all “direct” linear solver operations listed in §10.1:

- `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`

## 10.4 The SUNLinSol\_Dense Module

The SUNLinSol\_Dense implementation of the SUNLinearSolver class 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).



### 10.4.1 SUNLinSol\_Dense Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_dense.h`. The `SUNLinSol_Dense` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsoldense` module library.

The module `SUNLinSol_Dense` provides the following user-callable constructor routine:

*SUNLinearSolver* **SUNLinSol\_Dense**(*N\_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)

This function creates and allocates memory for a dense `SUNLinearSolver`.

**Arguments:**

- y – vector used to determine the linear system size.
- A – matrix used to assess compatibility.
- sunctx – the *SUNContext* object (see §4.2)

**Return value:**

New `SUNLinSol_Dense` object, or NULL if either A or y are incompatible.

**Notes:**

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.

### 10.4.2 SUNLinSol\_Dense Description

The `SUNLinSol_Dense` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Dense {  
    sunindextype N;  
    sunindextype *pivots;  
    sunindextype last_flag;  
};
```

These entries of the *content* field contain the following information:

- N - size of the linear system,
- pivots - index array for partial pivoting in LU factorization,
- last\_flag - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The “setup” call performs an *LU* factorization with partial (row) pivoting ( $\mathcal{O}(N^3)$  cost),  $PA = LU$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular matrix with 1’s on the diagonal, and  $U$  is an upper triangular matrix. This factorization is stored in-place on the input `SUNMATRIX_DENSE` object  $A$ , with pivoting information encoding  $P$  stored in the `pivots` array.
- The “solve” call performs pivoting and forward and backward substitution using the stored `pivots` array and the *LU* factors held in the `SUNMATRIX_DENSE` object ( $\mathcal{O}(N^2)$  cost).

The `SUNLinSol_Dense` module defines dense implementations of all “direct” linear solver operations listed in §10.1:

- `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`

## 10.5 The SUNLinSol\_KLU Module

The `SUNLinSol_KLU` implementation of the `SUNLinearSolver` class 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`).

### 10.5.1 SUNLinSol\_KLU Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_klu.h`. The installed module library to link to is `libsundials_sunlinsolklu.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module `SUNLinSol_KLU` provides the following additional user-callable routines:

*SUNLinearSolver* **`SUNLinSol_KLU`**(*N\_Vector* `y`, *SUNMatrix* `A`, *SUNContext* `sunctx`)

This constructor function creates and allocates memory for a `SUNLinSol_KLU` object.

#### Arguments:

- `y` – vector used to determine the linear system size.
- `A` – matrix used to assess compatibility.
- `sunctx` – the *SUNContext* object (see §4.2)

#### Return value:

New `SUNLinSol_KLU` object, or `NULL` if either `A` or `y` are incompatible.

#### Notes:

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.

*SUNErrCode* **`SUNLinSol_KLUReInit`**(*SUNLinearSolver* `S`, *SUNMatrix* `A`, *sunindextype* `nnz`, `int` `reinit_type`)

This function reinitializes memory and flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeros has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

#### Arguments:

- `S` – existing `SUNLinSol_KLU` object to reinitialize.
- `A` – sparse `SUNMatrix` matrix (with updated structure) to use for reinitialization.
- `nnz` – maximum number of nonzeros expected for Jacobian matrix.

- *reinit\_type* – 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).

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine assumes no other changes to solver use are necessary.

*SUNErrCode* **SUNLinSol\_KLUSetOrdering**(*SUNLinearSolver* S, int ordering\_choice)

This function sets the ordering used by KLU for reducing fill in the linear solve.

**Arguments:**

- *S* – existing `SUNLinSol_KLU` object to update.
- *ordering\_choice* – type of ordering to use, options are:
  0. AMD,
  1. COLAMD, and
  2. the natural ordering.

The default is 1 for COLAMD.

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by `SUNLinSolSetOptions()` when using the key “LSid.ordering”.

*sun\_klu\_symbolic* \***SUNLinSol\_KLUGetSymbolic**(*SUNLinearSolver* S)

This function returns a pointer to the KLU symbolic factorization stored in the `SUNLinSol_KLU` content structure.

type **sun\_klu\_symbolic**

This type is an alias that depends on the SUNDIALS index size, see *sunindextype* and *SUNDIALS\_INDEX\_SIZE*. It is equivalent to:

- `klu_symbolic` when SUNDIALS is compiled with 32-bit indices
- `klu_l_symbolic` when SUNDIALS is compiled with 64-bit indices

*sun\_klu\_numeric* \***SUNLinSol\_KLUGetNumeric**(*SUNLinearSolver* S)

This function returns a pointer to the KLU numeric factorization stored in the `SUNLinSol_KLU` content structure.

type **sun\_klu\_numeric**

This type is an alias that depends on the SUNDIALS index size, see *sunindextype* and *SUNDIALS\_INDEX\_SIZE*. It is equivalent to:

- `klu_numeric` when SUNDIALS is compiled with 32-bit indices
- `klu_l_numeric` when SUNDIALS is compiled with 64-bit indices

`sun_klu_common *SUNLinSol_KLUGetCommon(SUNLinearSolver S)`

This function returns a pointer to the KLU common structure stored in the SUNLinSol\_KLU content structure.

type **sun\_klu\_common**

This type is an alias that depends on the SUNDIALS index size, see [sunindextype](#) and [SUNDIALS\\_INDEX\\_SIZE](#). It is equivalent to:

- `klu_common` when SUNDIALS is compiled with 32-bit indices
- `klu_l_common` when SUNDIALS is compiled with 64-bit indices

## 10.5.2 SUNLinSol\_KLU Description

The SUNLinSol\_KLU module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_KLU {
    int          last_flag;
    int          first_factorize;
    sun_klu_symbolic *symbolic;
    sun_klu_numeric *numeric;
    sun_klu_common common;
    sunindextype (*klu_solver)(sun_klu_symbolic*, sun_klu_numeric*,
                               sunindextype, sunindextype,
                               double*, sun_klu_common*);
};
```

These entries of the *content* field contain the following information:

- `last_flag` - last error return flag from internal function evaluations,
- `first_factorize` - flag indicating whether the factorization has ever been performed,
- `symbolic` - KLU storage structure for symbolic factorization components, with underlying type `klu_symbolic` or `klu_l_symbolic`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,
- `numeric` - KLU storage structure for numeric factorization components, with underlying type `klu_numeric` or `klu_l_numeric`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,
- `common` - storage structure for common KLU solver components, with underlying type `klu_common` or `klu_l_common`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,
- `klu_solver` – pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix, and on whether SUNDIALS was installed with 32-bit or 64-bit indices).

The SUNLinSol\_KLU module is a SUNLinearSolver wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis and collaborators ([4, 32]). 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 §17.3.22 for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have [sunrealtype](#) set to either `extended` or `single` (see §4.1 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  $\varepsilon$  is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine `SUNKLUREInit`, that can be called by the user to force a full refactorization at the next “setup” call.
- The “solve” call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The SUNLinSol\_KLU module defines implementations of all “direct” linear solver operations listed in §10.1:

- `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`

## 10.6 The SUNLinSol\_LapackBand Module

The SUNLinSol\_LapackBand implementation of the `SUNLinearSolver` class 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

### 10.6.1 SUNLinSol\_LapackBand Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_lapackband.h`. The installed module library to link to is `libsundials_sunlinsollapackband.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

The module `SUNLinSol_LapackBand` provides the following user-callable routine:

*SUNLinearSolver* **SUNLinSol\_LapackBand**(*N\_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)

This function creates and allocates memory for a LAPACK band *SUNLinearSolver*.

**Arguments:**

- y – vector used to determine the linear system size.
- A – matrix used to assess compatibility.
- *sunctx* – the *SUNContext* object (see §4.2)

**Return value:**

New *SUNLinSol\_LapackBand* object, or NULL if either A or y are incompatible.

**Notes:**

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.

## 10.6.2 SUNLinSol\_LapackBand Description

*SUNLinSol\_LapackBand* module defines the *content* field of a *SUNLinearSolver* to be the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

- N - size of the linear system,
- *pivots* - index array for partial pivoting in LU factorization,
- *last\_flag* - last error return flag from internal function evaluations.

The *SUNLinSol\_LapackBand* module is a *SUNLinearSolver* wrapper for the LAPACK band matrix factorization and solve routines, \*GBTRF and \*GBTRS, where \* is either D or S, depending on whether SUNDIALS was configured to have *sunrealtype* set to double or single, respectively (see §4.1 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 §17.3.25 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 *sunrealtype*. 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 an *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  $m_l$ , then the upper triangular factor  $U$  can have upper bandwidth as big as  $\text{smu} = \text{MIN}(N-1, \mu+m_l)$ . The lower triangular factor  $L$  has lower bandwidth  $m_l$ .

The `SUNLinSol_LapackBand` module defines band implementations of all “direct” linear solver operations listed in §10.1:

- `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`

## 10.7 The `SUNLinSol_LapackDense` Module

The `SUNLinSol_LapackDense` implementation of the `SUNLinearSolver` class 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`).

### 10.7.1 `SUNLinSol_LapackDense` Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_lapackdense.h`. The installed module library to link to is `libsundials_sunlinsollapackdense.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

The module `SUNLinSol_LapackDense` provides the following additional user-callable constructor routine:

*SUNLinearSolver* **`SUNLinSol_LapackDense`**(*N\_Vector* `y`, *SUNMatrix* `A`, *SUNContext* `sunctx`)

This function creates and allocates memory for a LAPACK dense `SUNLinearSolver`.

**Arguments:**

- `y` – vector used to determine the linear system size.
- `A` – matrix used to assess compatibility.
- `sunctx` – the *SUNContext* object (see §4.2)

**Return value:**

New `SUNLinSol_LapackDense` object, or `NULL` if either `A` or `y` are incompatible.

**Notes:**

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.

## 10.7.2 SUNLinSol\_LapackDense Description

The SUNLinSol\_LapackDense module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
    sunindextype N;
    sunindextype *pivots;
    sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

- *N* - size of the linear system,
- *pivots* - index array for partial pivoting in LU factorization,
- *last\_flag* - last error return flag from internal function evaluations.

The SUNLinSol\_LapackDense module is a SUNLinearSolver wrapper for the LAPACK dense matrix factorization and solve routines, \*GETRF and \*GETRS, where \* is either D or S, depending on whether SUNDIALS was configured to have *sunrealtype* set to double or single, respectively (see §4.1 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 §17.3.25 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 *sunrealtype*. 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 an *LU* factorization with partial (row) pivoting ( $\mathcal{O}(N^3)$  cost),  $PA = LU$ , where *P* is a permutation matrix, *L* is a lower triangular matrix with 1’s on the diagonal, and *U* is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX\_DENSE object *A*, with pivoting information encoding *P* stored in the *pivots* array.
- The “solve” call performs pivoting and forward and backward substitution using the stored *pivots* array and the *LU* factors held in the SUNMATRIX\_DENSE object ( $\mathcal{O}(N^2)$  cost).

The SUNLinSol\_LapackDense module defines dense implementations of all “direct” linear solver operations listed in §10.1:

- 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



## 10.8 The SUNLinSol\_MagmaDense Module

The `SUNLinearSolver_MagmaDense` implementation of the `SUNLinearSolver` class is designed to be used with the `SUNMATRIX_MAGMADENSE` matrix, and a GPU-enabled vector. The header file to include when using this module is `sunlinsol/sunlinsol_magmadense.h`. The installed library to link to is `libsundials_sunlinsolmagmadense.lib` where `lib` is typically `.so` for shared libraries and `.a` for static libraries.

### Warning

The `SUNLinearSolver_MagmaDense` module is experimental and subject to change.

### 10.8.1 SUNLinearSolver\_MagmaDense Description

The `SUNLinearSolver_MagmaDense` implementation provides an interface to the dense LU and dense batched LU methods in the [MAGMA](#) linear algebra library [117]. The batched LU methods are leveraged when solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix} x_j = b_j.$$

### 10.8.2 SUNLinearSolver\_MagmaDense Functions

The `SUNLinearSolver_MagmaDense` module defines implementations of all “direct” linear solver operations listed in §10.1:

- `SUNLinSolGetType_MagmaDense`
- `SUNLinSolInitialize_MagmaDense`
- `SUNLinSolSetup_MagmaDense`
- `SUNLinSolSolve_MagmaDense`
- `SUNLinSolLastFlag_MagmaDense`
- `SUNLinSolFree_MagmaDense`

In addition, the module provides the following user-callable routines:

*SUNLinearSolver* **SUNLinSol\_MagmaDense**(*N\_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)

This constructor function creates and allocates memory for a `SUNLinearSolver` object.

#### Arguments:

- y – a vector for checking compatibility with the solver.
- A – a `SUNMATRIX_MAGMADENSE` matrix for checking compatibility with the solver.
- *sunctx* – the *SUNContext* object (see §4.2)

#### Return value:

If successful, a `SUNLinearSolver` object. If either A or y are incompatible then this routine will return NULL. This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.



*SUNErrCode* **SUNLinSol\_MagmaDense\_SetAsync**(*SUNLinearSolver* LS, *sunboolean*type onoff)

This function can be used to toggle the linear solver between asynchronous and synchronous modes. In asynchronous mode (default), SUNLinearSolver operations are asynchronous with respect to the host. In synchronous mode, the host and GPU device are synchronized prior to the operation returning.

**Arguments:**

- *LS* – a SUNLinSol\_MagmaDense object
- *onoff* – 0 for synchronous mode or 1 for asynchronous mode (default 1)

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by *SUNLinSolSetOptions()* when using the key “LSid.async”.

### 10.8.3 SUNLinearSolver\_MagmaDense Content

The SUNLinearSolver\_MagmaDense module defines the object *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_MagmaDense {
    int                last_flag;
    sunboolean         async;
    sunindextype       N;
    SUNMemory          pivots;
    SUNMemory          pivotsarr;
    SUNMemory          dpivotsarr;
    SUNMemory          infoarr;
    SUNMemory          rhsarr;
    SUNMemoryHelper    memhelp;
    magma_queue_t      q;
};
```

## 10.9 The SUNLinSol\_OneMklDense Module

The SUNLinearSolver\_OneMklDense implementation of the SUNLinearSolver class interfaces to the direct linear solvers from the [Intel oneAPI Math Kernel Library \(oneMKL\)](#) for solving dense systems or block-diagonal systems with dense blocks. This linear solver is best paired with the SUNMatrix\_OneMklDense matrix.

The header file to include when using this class is `sunlinsol/sunlinsol_onemkldense.h`. The installed library to link to is `libsundials_sunlinsolonemkldense.lib` where `lib` is typically `.so` for shared libraries and `.a` for static libraries.

**Warning**

The SUNLinearSolver\_OneMklDense class is experimental and subject to change.

### 10.9.1 SUNLinearSolver\_OneMklDense Functions

The `SUNLinearSolver_OneMklDense` class defines implementations of all “direct” linear solver operations listed in §10.1:

- `SUNLinSolGetType_OneMklDense` – returns `SUNLINEARSOLVER_ONEMKLDENSE`
- `SUNLinSolInitialize_OneMklDense`
- `SUNLinSolSetup_OneMklDense`
- `SUNLinSolSolve_OneMklDense`
- `SUNLinSolLastFlag_OneMklDense`
- `SUNLinSolFree_OneMklDense`

In addition, the class provides the following user-callable routines:

*SUNLinearSolver* **`SUNLinSol_OneMklDense`**(*N\_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)

This constructor function creates and allocates memory for a `SUNLinearSolver` object.

**Arguments:**

- y – a vector for checking compatibility with the solver.
- A – a `SUNMatrix_OneMklDense` matrix for checking compatibility with the solver.
- *sunctx* – the *SUNContext* object (see §4.2)

**Return value:**

If successful, a `SUNLinearSolver` object. If either A or y are incompatible then this routine will return NULL. This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

### 10.9.2 SUNLinearSolver\_OneMklDense Usage Notes

**Warning**

The `SUNLinearSolver_OneMklDense` class only supports 64-bit indexing, thus SUNDIALS must be built for 64-bit indexing to use this class.

When using the `SUNLinearSolver_OneMklDense` class with a SUNDIALS package (e.g. CVODE), the queue given to the matrix is also used for the linear solver.

## 10.10 The SUNLinSol\_PCG Module

The `SUNLinSol_PCG` implementation of the `SUNLinearSolver` class performs the PCG (Preconditioned Conjugate Gradient [58]) method; this is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation 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

$$\tilde{A}\tilde{x} = \tilde{b} \quad (10.4)$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \quad (10.5)$$

The scaling matrix must be chosen so that the vectors  $SP^{-1}b$  and  $S^{-1}Px$  have dimensionless components.

The stopping test for the PCG iterations is on the L2 norm of the scaled preconditioned residual:

$$\begin{aligned} &\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta \\ \Leftrightarrow & \|SP^{-1}b - SP^{-1}Ax\|_2 < \delta \\ \Leftrightarrow & \|P^{-1}b - P^{-1}Ax\|_S < \delta \end{aligned}$$

where  $\|v\|_S = \sqrt{v^T S^T S v}$ , with an input tolerance  $\delta$ .

### 10.10.1 SUNLinSol\_PCG Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_pcg.h`. The `SUNLinSol_PCG` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolpcg` module library.

The module `SUNLinSol_PCG` provides the following user-callable routines:

*SUNLinearSolver* **SUNLinSol\_PCG**(*N\_Vector* y, int pretype, int maxl, *SUNContext* sunctx)

This constructor function creates and allocates memory for a PCG `SUNLinearSolver`.

#### Arguments:

- *y* – a template vector.
- *pretype* – a flag indicating the type of preconditioning to use:
  - `SUN_PREC_NONE`
  - `SUN_PREC_LEFT`
  - `SUN_PREC_RIGHT`
  - `SUN_PREC_BOTH`
- *maxl* – the maximum number of linear iterations to allow.
- *sunctx* – the *SUNContext* object (see §4.2)

**Return value:**

If successful, a `SUNLinearSolver` object. If either `y` is incompatible then this routine will return `NULL`.

**Notes:**

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).

A `maxl` argument that is  $\leq 0$  will result in the default value (5).

Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the `pre-type` inputs `SUN_PREC_LEFT`, `SUN_PREC_RIGHT`, or `SUN_PREC_BOTH` 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.

*SUNErrCode* `SUNLinSol_PCGSetPrecType(SUNLinearSolver S, int pretype)`

This function updates the flag indicating use of preconditioning.

**Arguments:**

- `S` – `SUNLinSol_PCG` object to update.
- `pretype` – a flag indicating the type of preconditioning to use:
  - `SUN_PREC_NONE`
  - `SUN_PREC_LEFT`
  - `SUN_PREC_RIGHT`
  - `SUN_PREC_BOTH`

**Return value:**

- A *SUNErrCode*

**Notes:**

As above, any one of the input values, `SUN_PREC_LEFT`, `SUN_PREC_RIGHT`, or `SUN_PREC_BOTH` will enable preconditioning; `SUN_PREC_NONE` disables preconditioning.

This routine will be called by `SUNLinSolSetOptions()` when using the key “`LSid.prec_type`”.

*SUNErrCode* `SUNLinSol_PCGSetMaxl(SUNLinearSolver S, int maxl)`

This function updates the number of linear solver iterations to allow.

**Arguments:**

- `S` – `SUNLinSol_PCG` object to update.
- `maxl` – maximum number of linear iterations to allow. Any non-positive input will result in the default value (5).

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by `SUNLinSolSetOptions()` when using the key “`LSid.maxl`”.

### 10.10.2 SUNLinSol\_PCG Description

The SUNLinSol\_PCG module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_PCG {
    int maxl;
    int pretype;
    sunboolean_t zeroguess;
    int numiters;
    sunreal_t resnorm;
    int last_flag;
    SUNATimesFn ATimes;
    void* ATData;
    SUNPSetupFn Psetup;
    SUNPSolveFn 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 SUNLinSol\_PCG module defines implementations of all “iterative” linear solver operations listed in §10.1:

- 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.
- SUNLinSolSetZeroGuess\_PCG – note the solver assumes a non-zero guess by default and the zero guess flag is reset to SUNFALSE after each call to SUNLinSolSolve\_PCG.
- SUNLinSolSetup\_PCG
- SUNLinSolSolve\_PCG
- SUNLinSolNumIters\_PCG
- SUNLinSolResNorm\_PCG
- SUNLinSolResid\_PCG
- SUNLinSolLastFlag\_PCG
- SUNLinSolSpace\_PCG
- SUNLinSolFree\_PCG

## 10.11 The SUNLinSol\_SPBCGS Module

The SUNLinSol\_SPBCGS implementation of the SUNLinearSolver class performs a Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [124] method; this is an iterative linear solver that is designed to be compatible with any N\_Vector implementation 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.

### 10.11.1 SUNLinSol\_SPBCGS Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_spbcgs.h`. The SUNLinSol\_SPBCGS module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspbcgs` module library.

The module SUNLinSol\_SPBCGS provides the following user-callable routines:

*SUNLinearSolver* **SUNLinSol\_SPBCGS**(N\_Vector y, int pretype, int maxl, *SUNContext* sunctx)

This constructor function creates and allocates memory for a SPBCGS SUNLinearSolver.

**Arguments:**

- y – a template vector.

- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH
- *maxl* – the maximum number of linear iterations to allow.
- *sunctx* – the [SUNContext](#) object (see §4.2)

**Return value:**

If successful, a [SUNLinearSolver](#) object. If either *y* is incompatible then this routine will return NULL.

**Notes:**

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).

A *maxl* argument that is  $\leq 0$  will result in the default value (5).

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.

With [SUN\\_PREC\\_RIGHT](#) or [SUN\\_PREC\\_BOTH](#) the initial guess must be zero (use [SUNLinSolSetZeroGuess\(\)](#) to indicate the initial guess is zero).

[SUNErrCode](#) [SUNLinSol\\_SPBCGSSetPrecType](#)([SUNLinearSolver](#) S, int pretype)

This function updates the flag indicating use of preconditioning.

**Arguments:**

- *S* – [SUNLinSol\\_SPBCGS](#) object to update.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH

**Return value:**

- A [SUNErrCode](#)

**Notes:**

This routine will be called by [SUNLinSolSetOptions\(\)](#) when using the key “LSid.prec\_type”.

[SUNErrCode](#) [SUNLinSol\\_SPBCGSsetMaxl](#)([SUNLinearSolver](#) S, int maxl)

This function updates the number of linear solver iterations to allow.

**Arguments:**

- *S* – [SUNLinSol\\_SPBCGS](#) object to update.
- *maxl* – maximum number of linear iterations to allow. Any non-positive input will result in the default value (5).

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by *SUNLinSolSetOptions()* when using the key “LSid.maxl”.

### 10.11.2 SUNLinSol\_SPBCGS Description

The SUNLinSol\_SPBCGS module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SPBCGS {  
    int maxl;  
    int pretype;  
    sunbooleantype zeroguess;  
    int numiters;  
    sunrealtype resnorm;  
    int last_flag;  
    SUNATimesFn ATimes;  
    void* ATData;  
    SUNPSetupFn Psetup;  
    SUNPSolveFn 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 SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the SPBCGS iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The `SUNLinSol_SPBCGS` module defines implementations of all “iterative” linear solver operations listed in §10.1:

- `SUNLinSolGetType_SPBCGS`
- `SUNLinSolInitialize_SPBCGS`
- `SUNLinSolSetATimes_SPBCGS`
- `SUNLinSolSetPreconditioner_SPBCGS`
- `SUNLinSolSetScalingVectors_SPBCGS`
- `SUNLinSolSetZeroGuess_SPBCGS` – note the solver assumes a non-zero guess by default and the zero guess flag is reset to `SUNFALSE` after each call to `SUNLinSolSolve_SPBCGS`.
- `SUNLinSolSetup_SPBCGS`
- `SUNLinSolSolve_SPBCGS`
- `SUNLinSolNumIters_SPBCGS`
- `SUNLinSolResNorm_SPBCGS`
- `SUNLinSolResid_SPBCGS`
- `SUNLinSolLastFlag_SPBCGS`
- `SUNLinSolSpace_SPBCGS`
- `SUNLinSolFree_SPBCGS`

## 10.12 The `SUNLinSol_SPFGMR` Module

The `SUNLinSol_SPFGMR` implementation of the `SUNLinearSolver` class performs a Scaled, Preconditioned, Flexible, Generalized Minimum Residual [91] method; this is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation 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).

### 10.12.1 SUNLinSol\_SPFGMR Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_spfgmr.h`. The `SUNLinSol_SPFGMR` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspfgmr` module library.

The module `SUNLinSol_SPFGMR` provides the following user-callable routines:

*SUNLinearSolver* **SUNLinSol\_SPFGMR**(*N\_Vector* y, int pretype, int maxl, *SUNContext* sunctx)

This constructor function creates and allocates memory for a SPFGMR `SUNLinearSolver`.

**Arguments:**

- y – a template vector.
- pretype – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH
- maxl – the number of Krylov basis vectors to use.
- sunctx – the *SUNContext* object (see §4.2)

**Return value:**

If successful, a `SUNLinearSolver` object. If either y is incompatible then this routine will return NULL.

**Notes:**

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).

A `maxl` argument that is  $\leq 0$  will result in the default value (5).

Since the FGMRES algorithm is designed to only support right preconditioning, then any of the `pretype` inputs `SUN_PREC_LEFT`, `SUN_PREC_RIGHT`, or `SUN_PREC_BOTH` will result in use of `SUN_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.

*SUNErrCode* **SUNLinSol\_SPFGMRSetPrecType**(*SUNLinearSolver* S, int pretype)

This function updates the flag indicating use of preconditioning.

**Arguments:**

- S – `SUNLinSol_SPFGMR` object to update.
- pretype – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH

**Return value:**

- A *SUNErrCode*

**Notes:**

Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs `SUN_PREC_LEFT`, `SUN_PREC_RIGHT`, or `SUN_PREC_BOTH` will result in use of `SUN_PREC_RIGHT`; any other integer input will result in the default (no preconditioning).

This routine will be called by `SUNLinSolSetOptions()` when using the key “LSid.prec\_type”.

*SUNErrCode* **SUNLinSol\_SPFGMRSetGSType**(*SUNLinearSolver* S, int gstype)

This function sets the type of Gram-Schmidt orthogonalization to use.

**Arguments:**

- *S* – `SUNLinSol_SPFGMR` object to update.
- *gstype* – a flag indicating the type of orthogonalization to use:
  - `SUN_MODIFIED_GS`
  - `SUN_CLASSICAL_GS`

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by `SUNLinSolSetOptions()` when using the key “LSid.gs\_type”.

*SUNErrCode* **SUNLinSol\_SPFGMRSetMaxRestarts**(*SUNLinearSolver* S, int maxrs)

This function sets the number of FGMRES restarts to allow.

**Arguments:**

- *S* – `SUNLinSol_SPFGMR` object to update.
- *maxrs* – maximum number of restarts to allow. A negative input will result in the default of 0.

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by `SUNLinSolSetOptions()` when using the key “LSid.max\_restarts”.

### 10.12.2 SUNLinSol\_SPFGMR Description

The `SUNLinSol_SPFGMR` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    sunbooleantype zeroguess;
    int numiters;
    sunrealtype resnorm;
    int last_flag;
    SUNATimesFn ATimes;
    void* ATData;
    SUNPSsetupFn Psetup;
    SUNPSolveFn Psolve;
}
```

(continues on next page)

(continued from previous page)

```

void* PData;
N_Vector s1;
N_Vector s2;
N_Vector *V;
N_Vector *Z;
sunrealtype **Hes;
sunrealtype *givens;
N_Vector xcor;
sunrealtype *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  $v_1, \dots, v_{\text{maxl}+1}$ , stored in `V[0]`,  $\dots$ , `V[maxl]`. Each  $v_i$  is a vector of type `N_Vector`,
- `Z` - the array of preconditioned Krylov basis vectors  $z_1, \dots, z_{\text{maxl}+1}$ , stored in `Z[0]`,  $\dots$ , `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  $F_0, F_1, \dots, F_j$ , 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] = c0`, `givens[1] = s0`, `givens[2] = c1`, `givens[3] = s1`, ..., `givens[2j] = cj`, `givens[2j+1] = sj`,

- `xcor` - a vector which holds the scaled, preconditioned correction to the initial guess,
- `yg` - a length  $(\text{maxl} + 1)$  array of `sunrealtype` 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 `SUNLinSol_SPFGMR` module defines implementations of all “iterative” linear solver operations listed in §10.1:

- `SUNLinSolGetType_SPFGMR`
- `SUNLinSolInitialize_SPFGMR`
- `SUNLinSolSetATimes_SPFGMR`
- `SUNLinSolSetPreconditioner_SPFGMR`
- `SUNLinSolSetScalingVectors_SPFGMR`
- `SUNLinSolSetZeroGuess_SPFGMR` – note the solver assumes a non-zero guess by default and the zero guess flag is reset to `SUNFALSE` after each call to `SUNLinSolSolve_SPFGMR`.
- `SUNLinSolSetup_SPFGMR`
- `SUNLinSolSolve_SPFGMR`
- `SUNLinSolNumIters_SPFGMR`
- `SUNLinSolResNorm_SPFGMR`
- `SUNLinSolResid_SPFGMR`
- `SUNLinSolLastFlag_SPFGMR`
- `SUNLinSolSpace_SPFGMR`
- `SUNLinSolFree_SPFGMR`

## 10.13 The SUNLinSol\_SPGMR Module

The `SUNLinSol_SPGMR` implementation of the `SUNLinearSolver` class performs a Scaled, Preconditioned, Generalized Minimum Residual [92] method; this is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation that supports a minimal subset of operations (`N_VClone()`, `N_VDotProd()`, `N_VScale()`, `N_VLinearSum()`, `N_VProd()`, `N_VConst()`, `N_VDiv()`, and `N_VDestroy()`).

### 10.13.1 SUNLinSol\_SPGMR Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_spgmr.h`. The `SUNLinSol_SPGMR` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspgmr` module library.

The module `SUNLinSol_SPGMR` provides the following user-callable routines:

*SUNLinearSolver* **SUNLinSol\_SPGMR**(*N\_Vector* y, int pretype, int maxl, *SUNContext* sunctx)

This constructor function creates and allocates memory for a SPGMR `SUNLinearSolver`.

**Arguments:**

- y – a template vector.
- pretype – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH
- maxl – the number of Krylov basis vectors to use.

**Return value:**

If successful, a `SUNLinearSolver` object. If either y is incompatible then this routine will return NULL.

**Notes:**

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).

A `maxl` argument that is  $\leq 0$  will result in the default value (5).

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.

*SUNErrCode* **SUNLinSol\_SPGMRSetPrecType**(*SUNLinearSolver* S, int pretype)

This function updates the flag indicating use of preconditioning.

**Arguments:**

- S – `SUNLinSol_SPGMR` object to update.
- pretype – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by `SUNLinSolSetOptions()` when using the key “LSid.prec\_type”.

*SUNErrCode* **SUNLinSol\_SPGMRSetGSType**(*SUNLinearSolver* S, int gstype)

This function sets the type of Gram-Schmidt orthogonalization to use.

**Arguments:**

- *S* – SUNLinSol\_SPGMR object to update.
- *gstype* – a flag indicating the type of orthogonalization to use:
  - SUN\_MODIFIED\_GS
  - SUN\_CLASSICAL\_GS

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by *SUNLinSolSetOptions()* when using the key “LSid.gs\_type”.

*SUNErrCode* **SUNLinSol\_SPGMRSetMaxRestarts**(*SUNLinearSolver* S, int maxrs)

This function sets the number of GMRES restarts to allow.

**Arguments:**

- *S* – SUNLinSol\_SPGMR object to update.
- *maxrs* – maximum number of restarts to allow. A negative input will result in the default of 0.

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by *SUNLinSolSetOptions()* when using the key “LSid.max\_restarts”.

### 10.13.2 SUNLinSol\_SPGMR Description

The SUNLinSol\_SPGMR module defines the *content* field of a *SUNLinearSolver* to be the following structure:

```
struct _SUNLinearSolverContent_SPGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    sunboolean_t zeroguess;
    int numiters;
    sunreal_t resnorm;
    int last_flag;
    SUNATimesFn ATimes;
    void* ATData;
    SUNPSetupFn Psetup;
    SUNPSolveFn Psolve;
    void* PData;
    N_Vector s1;
    N_Vector s2;
    N_Vector *V;
    sunreal_t **Hes;
    sunreal_t *givens;
```

(continues on next page)

(continued from previous page)

```

N_Vector xcor;
sunrealtype *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  $v_1, \dots, v_{\text{maxl}+1}$ , stored in **V[0]**,  $\dots$  **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  $F_0, F_1, \dots, F_j$ , 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]** =  $s_1$ ,  $\dots$ , **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 **sunrealtype** 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 `SUNLinSol_SPGMR` module defines implementations of all “iterative” linear solver operations listed in §10.1:

- `SUNLinSolGetType_SPGMR`
- `SUNLinSolInitialize_SPGMR`
- `SUNLinSolSetATimes_SPGMR`
- `SUNLinSolSetPreconditioner_SPGMR`
- `SUNLinSolSetScalingVectors_SPGMR`
- `SUNLinSolSetZeroGuess_SPGMR` – note the solver assumes a non-zero guess by default and the zero guess flag is reset to `SUNFALSE` after each call to `SUNLinSolSolve_SPGMR`.
- `SUNLinSolSetup_SPGMR`
- `SUNLinSolSolve_SPGMR`
- `SUNLinSolNumIters_SPGMR`
- `SUNLinSolResNorm_SPGMR`
- `SUNLinSolResid_SPGMR`
- `SUNLinSolLastFlag_SPGMR`
- `SUNLinSolSpace_SPGMR`
- `SUNLinSolFree_SPGMR`

## 10.14 The `SUNLinSol_SPTFQMR` Module

The `SUNLinSol_SPTFQMR` implementation of the `SUNLinearSolver` class performs a Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [47] method; this is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation 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.

### 10.14.1 SUNLinSol\_SPTFQMR Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_sptfqmr.h`. The `SUNLinSol_SPTFQMR` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolsptfqmr` module library.

The module `SUNLinSol_SPTFQMR` provides the following user-callable routines:

*SUNLinearSolver* **SUNLinSol\_SPTFQMR**(*N\_Vector* y, int pretype, int maxl, *SUNContext* suctx)

This constructor function creates and allocates memory for a SPTFQMR `SUNLinearSolver`.

**Arguments:**

- y – a template vector.
- pretype – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH
- maxl – the number of Krylov basis vectors to use.
- suctx – the *SUNContext* object (see §4.2)

**Return value:**

If successful, a `SUNLinearSolver` object. If either y is incompatible then this routine will return NULL.

**Notes:**

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).

A `maxl` argument that is  $\leq 0$  will result in the default value (5).

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.

With `SUN_PREC_RIGHT` or `SUN_PREC_BOTH` the initial guess must be zero (use *SUNLinSolSetZeroGuess()* to indicate the initial guess is zero).

*SUNErrCode* **SUNLinSol\_SPTFQMRSetPrecType**(*SUNLinearSolver* S, int pretype)

This function updates the flag indicating use of preconditioning.

**Arguments:**

- S – `SUNLinSol_SPGMR` object to update.
- pretype – a flag indicating the type of preconditioning to use:
  - SUN\_PREC\_NONE
  - SUN\_PREC\_LEFT
  - SUN\_PREC\_RIGHT
  - SUN\_PREC\_BOTH

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by `SUNLinSolSetOptions()` when using the key “LSid.prec\_type”.

*SUNErrCode* `SUNLinSol_SPTFQMRSetMaxl(SUNLinearSolver S, int maxl)`

This function updates the number of linear solver iterations to allow.

**Arguments:**

- *S* – SUNLinSol\_SPTFQMR object to update.
- *maxl* – maximum number of linear iterations to allow. Any non-positive input will result in the default value (5).

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by `SUNLinSolSetOptions()` when using the key “LSid.maxl”.

### 10.14.2 SUNLinSol\_SPTFQMR Description

The SUNLinSol\_SPTFQMR module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SPTFQMR {
    int maxl;
    int pretype;
    sunboolean_t zeroguess;
    int numiters;
    sunreal_t resnorm;
    int last_flag;
    SUNATimesFn ATimes;
    void* ATData;
    SUNPSolveFn Psetup;
    SUNPSolveFn 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 SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The `SUNLinSol_SPTFQMR` module defines implementations of all “iterative” linear solver operations listed in §10.1:

- `SUNLinSolGetType_SPTFQMR`
- `SUNLinSolInitialize_SPTFQMR`
- `SUNLinSolSetATimes_SPTFQMR`
- `SUNLinSolSetPreconditioner_SPTFQMR`
- `SUNLinSolSetScalingVectors_SPTFQMR`
- `SUNLinSolSetZeroGuess_SPTFQMR` – note the solver assumes a non-zero guess by default and the zero guess flag is reset to `SUNFALSE` after each call to `SUNLinSolSolve_SPTFQMR`.
- `SUNLinSolSetup_SPTFQMR`
- `SUNLinSolSolve_SPTFQMR`
- `SUNLinSolNumIters_SPTFQMR`
- `SUNLinSolResNorm_SPTFQMR`
- `SUNLinSolResid_SPTFQMR`
- `SUNLinSolLastFlag_SPTFQMR`

- `SUNLinSolSpace_SPTFQMR`
- `SUNLinSolFree_SPTFQMR`

## 10.15 The SUNLinSol\_SuperLUDIST Module

The `SUNLinSol_SuperLUDIST` implementation of the `SUNLinearSolver` class interfaces with the `SuperLU_DIST` library. This is designed to be used with the `SUNMatrix_SLUNRloc` *SUNMatrix*, and one of the serial, threaded or parallel `N_Vector` implementations (`NVECTOR_SERIAL`, `NVECTOR_OPENMP`, `NVECTOR_PTHREADS`, `NVECTOR_PARALLEL`, `NVECTOR_PARHYP`).

### 10.15.1 SUNLinSol\_SuperLUDIST Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_superludist.h`. The installed module library to link to is `libsundials_sunlinsolsuperludist.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

The module `SUNLinSol_SuperLUDIST` provides the following user-callable routines:

#### Warning

Starting with `SuperLU_DIST` version 6.3.0, some structures were renamed to have a prefix for the floating point type. The double precision API functions have the prefix 'd'. To maintain backwards compatibility with the un-prefixed types, SUNDIALS provides macros to these `SuperLU_DIST` types with an 'x' prefix that expand to the correct prefix. E.g., the SUNDIALS macro `xLUstruct_t` expands to `dLUstruct_t` or `LUstruct_t` based on the `SuperLU_DIST` version.

*SUNLinearSolver* **SUNLinSol\_SuperLUDIST**(*N\_Vector* y, SuperMatrix \*A, gridinfo\_t \*grid, xLUstruct\_t \*lu, xScalePermstruct\_t \*scaleperm, xSOLVEstruct\_t \*solve, SuperLUStat\_t \*stat, superlu\_dist\_options\_t \*options, *SUNContext* sunctx)

This constructor function creates and allocates memory for a `SUNLinSol_SuperLUDIST` object.

#### Arguments:

- y – a template vector.
- A – a template matrix
- grid, lu, scaleperm, solve, stat, options – `SuperLU_DIST` object pointers.
- sunctx – the *SUNContext* object (see §4.2)

#### Return value:

If successful, a `SUNLinearSolver` object; otherwise this routine will return `NULL`.

#### Notes:

This routine analyzes the input matrix and vector to determine the linear system size and to assess the compatibility with the `SuperLU_DIST` library.

This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the `SUNMatrix_SLUNRloc` matrix type and the `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, `NVECTOR_PTHREADS`, `NVECTOR_PARALLEL`, and `NVECTOR_PARHYP` vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

The `grid`, `lu`, `scaleperm`, `solve`, and `options` arguments are not checked and are passed directly to `SuperLU_DIST` routines.

Some struct members of the `options` argument are modified internally by the `SUNLinSol_SuperLUDIST` solver. Specifically, the member `Fact` is modified in the `setup` and `solve` routines.

*sunrealtype* **SUNLinSol\_SuperLUDIST\_GetBerr**(*SUNLinearSolver* LS)

This function returns the componentwise relative backward error of the computed solution. It takes one argument, the `SUNLinearSolver` object. The return type is `sunrealtype`.

`gridinfo_t` \***SUNLinSol\_SuperLUDIST\_GetGridinfo**(*SUNLinearSolver* LS)

This function returns a pointer to the `SuperLU_DIST` structure that contains the 2D process grid. It takes one argument, the `SUNLinearSolver` object.

`xLUstruct_t` \***SUNLinSol\_SuperLUDIST\_GetLUstruct**(*SUNLinearSolver* LS)

This function returns a pointer to the `SuperLU_DIST` structure that contains the distributed L and U structures. It takes one argument, the `SUNLinearSolver` object.

`superlu_dist_options_t` \***SUNLinSol\_SuperLUDIST\_GetSuperLUOptions**(*SUNLinearSolver* LS)

This function returns a pointer to the `SuperLU_DIST` structure that contains the options which control how the linear system is factorized and solved. It takes one argument, the `SUNLinearSolver` object.

`xScalePermstruct_t` \***SUNLinSol\_SuperLUDIST\_GetScalePermstruct**(*SUNLinearSolver* LS)

This function returns a pointer to the `SuperLU_DIST` structure that contains the vectors that describe the transformations done to the matrix A. It takes one argument, the `SUNLinearSolver` object.

`xSOLVEstruct_t` \***SUNLinSol\_SuperLUDIST\_GetSOLVEstruct**(*SUNLinearSolver* LS)

This function returns a pointer to the `SuperLU_DIST` structure that contains information for communication during the solution phase. It takes one argument the `SUNLinearSolver` object.

`SuperLUStat_t` \***SUNLinSol\_SuperLUDIST\_GetSuperLUStat**(*SUNLinearSolver* LS)

This function returns a pointer to the `SuperLU_DIST` structure that stores information about runtime and flop count. It takes one argument, the `SUNLinearSolver` object.

### 10.15.2 SUNLinSol\_SuperLUDIST Description

The `SUNLinSol_SuperLUDIST` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SuperLUDIST {
    sunboolean_t    first_factorize;
    int             last_flag;
    sunrealtype     berr;
    gridinfo_t      *grid;
    xLUstruct_t     *lu;
    superlu_dist_options_t *options;
    xScalePermstruct_t *scaleperm;
    xSOLVEstruct_t  *solve;
    SuperLUStat_t   *stat;
    sunindextype     N;
};
```

These entries of the *content* field contain the following information:

- `first_factorize` – flag indicating whether the factorization has ever been performed,
- `last_flag` – last error return flag from internal function evaluations,

- `berr` – the componentwise relative backward error of the computed solution,
- `grid` – pointer to the `SuperLU_DIST` structure that stores the 2D process grid
- `lu` – pointer to the `SuperLU_DIST` structure that stores the distributed L and U factors,
- `scaleperm` – pointer to the `SuperLU_DIST` structure that stores vectors describing the transformations done to the matrix A,
- `options` – pointer to the `SuperLU_DIST` structure which contains options that control how the linear system is factorized and solved,
- `solve` – pointer to the `SuperLU_DIST` solve structure,
- `stat` – pointer to the `SuperLU_DIST` structure that stores information about runtime and flop count,
- `N` – the number of equations in the system.

The `SUNLinSol_SuperLUDIST` module is a `SUNLinearSolver` adapter for the `SuperLU_DIST` sparse matrix factorization and solver library written by X. Sherry Li and collaborators [8, 51, 77, 78]. The package uses a SPMD parallel programming model and multithreading to enhance efficiency in distributed-memory parallel environments with multi-core nodes and possibly GPU accelerators. It uses MPI for communication, OpenMP for threading, and CUDA for GPU support. In order to use the `SUNLinSol_SuperLUDIST` interface to `SuperLU_DIST`, it is assumed that `SuperLU_DIST` has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with `SuperLU_DIST` (see §17.3.34 for details). Additionally, the wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to use single or extended precision. Moreover, since the `SuperLU_DIST` library may be installed to support either 32-bit or 64-bit integers, it is assumed that the `SuperLU_DIST` library is installed using the same integer size as SUNDIALS.

The `SuperLU_DIST` library provides many options to control how a linear system will be factorized and solved. These options may be set by a user on an instance of the `superlu_dist_options_t` struct, and then it may be provided as an argument to the `SUNLinSol_SuperLUDIST` constructor. The `SUNLinSol_SuperLUDIST` module will respect all options set except for `Fact` – this option is necessarily modified by the `SUNLinSol_SuperLUDIST` module in the setup and solve routines.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the `SUNLinSol_SuperLUDIST` module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it sets the `SuperLU_DIST` option `Fact` to `DOFACT` so that a subsequent call to the “solve” routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to solve the system.
- On subsequent calls to the “setup” routine, it sets the `SuperLU_DIST` option `Fact` to `SamePattern` so that a subsequent call to “solve” will perform factorization assuming the same sparsity pattern as prior, i.e. it will reuse the column permutation vector.
- If “setup” is called prior to the “solve” routine, then the “solve” routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to the sparse triangular solves, and, potentially, iterative refinement. If “setup” is not called prior, “solve” will skip to the triangular solve step. We note that in this solve `SuperLU_DIST` operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The `SUNLinSol_SuperLUDIST` module defines implementations of all “direct” linear solver operations listed in §10.1:

- `SUNLinSolGetType_SuperLUDIST`
- `SUNLinSolInitialize_SuperLUDIST` – this sets the `first_factorize` flag to 1 and resets the internal `SuperLU_DIST` statistics variables.
- `SUNLinSolSetup_SuperLUDIST` – this sets the appropriate `SuperLU_DIST` options so that a subsequent solve will perform a symbolic and numerical factorization before proceeding with the triangular solves

- `SUNLinSolSolve_SuperLUDIST` – this calls the `SuperLU_DIST` solve routine to perform factorization (if the setup routine was called prior) and then use the `$LU$` factors to solve the linear system.
- `SUNLinSolLastFlag_SuperLUDIST`
- `SUNLinSolSpace_SuperLUDIST` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the `SuperLU_DIST` documentation.
- `SUNLinSolFree_SuperLUDIST`

## 10.16 The SUNLinSol\_SuperLUMT Module

The `SUNLinSol_SuperLUMT` implementation of the `SUNLinearSolver` class interfaces with the `SuperLU_MT` library. This 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.

### 10.16.1 SUNLinSol\_SuperLUMT Usage

The header file to be included when using this module is `sunlinsol/sunlinsol.SuperLUMT.h`. The installed module library to link to is `libsundials_sunlinsolsuperlumt.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module `SUNLinSol_SuperLUMT` provides the following user-callable routines:

*SUNLinearSolver* **SUNLinSol\_SuperLUMT**(*N\_Vector* y, *SUNMatrix* A, int num\_threads, *SUNContext* sunctx)

This constructor function creates and allocates memory for a `SUNLinSol_SuperLUMT` object.

#### Arguments:

- y – a template vector.
- A – a template matrix
- *num\_threads* – desired number of threads (OpenMP or Pthreads, depending on how `SuperLU_MT` was installed) to use during the factorization steps.
- *sunctx* – the *SUNContext* object (see §4.2)

#### Return value:

If successful, a `SUNLinearSolver` object; otherwise this routine will return `NULL`.

#### Notes:

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.

The `num_threads` argument is not checked and is passed directly to `SuperLU_MT` routines.



*SUNErrCode* **SUNLinSol\_SuperLUMTSetOrdering**(*SUNLinearSolver* S, int ordering\_choice)

This function sets the ordering used by SuperLU\_MT for reducing fill in the linear solve.

**Arguments:**

- *S* – the SUNLinSol\_SuperLUMT object to update.
- *ordering\_choice*:
  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.

**Return value:**

- A *SUNErrCode*

**Notes:**

This routine will be called by *SUNLinSolSetOptions()* when using the key “LSid.ordering”.

## 10.16.2 SUNLinSol\_SuperLUMT Description

The SUNLinSol\_SuperLUMT module defines the *content* field of a *SUNLinearSolver* to be the following structure:

```
struct _SUNLinearSolverContent_SuperLUMT {
    int      last_flag;
    int      first_factorize;
    SuperMatrix *A, *AC, *L, *U, *B;
    Gstat_t   *Gstat;
    sunindextype *perm_r, *perm_c;
    sunindextype N;
    int      num_threads;
    sunrealtype 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 and collaborators [9, 34, 76]. 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 §17.3.35 for details). Additionally, this wrapper only supports single- and double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have `sunrealtype` set to `extended` (see §4.1 for details). Moreover, since the SuperLU\_MT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SuperLU\_MT library is installed using the same integer precision as the SUNDIALS `sunindextype` option.

The SuperLU\_MT library has a symbolic factorization routine that computes the permutation of the linear system matrix to reduce fill-in on subsequent *LU* factorizations (using COLAMD, minimal degree ordering on  $A^T * A$ , minimal degree ordering on  $A^T + A$ , or natural ordering). Of these ordering choices, the default value in the `SUNLinSol_SuperLUMT` module is the COLAMD ordering.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the `SUNLinSol_SuperLUMT` module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the “setup” routine, it skips the symbolic factorization, and only refactors the input matrix.
- The “solve” call performs pivoting and forward and backward substitution using the stored SuperLU\_MT data structures. We note that in this solve SuperLU\_MT operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The `SUNLinSol_SuperLUMT` module defines implementations of all “direct” linear solver operations listed in §10.1:

- `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`

## 10.17 The `SUNLinSol_cuSolverSp_batchQR` Module

The `SUNLinSol_cuSolverSp_batchQR` implementation of the `SUNLinearSolver` class is designed to be used with the `SUNMATRIX_CUSPARSE` matrix, and the `NVECTOR_CUDA` vector. The header file to include when using this module is `sunlinsol/sunlinsol_cusolversp_batchqr.h`. The installed library to link to is `libsundials_sunlinsolcusolversp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

**Warning**

The `SUNLinearSolver_cuSolverSp_batchQR` module is experimental and subject to change.

**10.17.1 SUNLinSol\_cuSolverSp\_batchQR description**

The `SUNLinearSolver_cuSolverSp_batchQR` implementation provides an interface to the batched sparse QR factorization method provided by the NVIDIA cuSOLVER library [6]. The module is designed for solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_n \end{bmatrix} x_j = b_j$$

where all block matrices  $\mathbf{A}_j$  share the same sparsity pattern. The matrix must be the `SUNMatrix.cuSparse`.

**10.17.2 SUNLinSol\_cuSolverSp\_batchQR functions**

The `SUNLinearSolver_cuSolverSp_batchQR` module defines implementations of all “direct” linear solver operations listed in §10.1:

- `SUNLinSolGetType_cuSolverSp_batchQR`
- `SUNLinSolInitialize_cuSolverSp_batchQR` – this sets the `first_factorize` flag to 1
- `SUNLinSolSetup_cuSolverSp_batchQR` – this always copies the relevant `SUNMATRIX_SPARSE` data to the GPU; if this is the first setup it will perform symbolic analysis on the system
- `SUNLinSolSolve_cuSolverSp_batchQR` – this calls the `cusolverSpXcsrqrsvBatched` routine to perform factorization
- `SUNLinSolLastFlag_cuSolverSp_batchQR`
- `SUNLinSolFree_cuSolverSp_batchQR`

In addition, the module provides the following user-callable routines:

*SUNLinearSolver* **`SUNLinSol_cuSolverSp_batchQR`**(*N\_Vector* y, *SUNMatrix* A, *cusolverHandle\_t* cusol, *SUNContext* sunctx)

The function `SUNLinSol_cuSolverSp_batchQR` creates and allocates memory for a `SUNLinearSolver` object.

**Arguments:**

- y – a vector for checking compatibility with the solver.
- A – a `SUNMATRIX_cuSparse` matrix for checking compatibility with the solver.
- *cusol* – `cuSolverSp` object to use.
- *sunctx* – the *SUNContext* object (see §4.2)

**Return value:**

If successful, a `SUNLinearSolver` object. If either A or y are incompatible then this routine will return `NULL`.

**Notes:**

This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the `SUNMATRIX_CUSPARSE` matrix type and the `NVECTOR_CUDA` vector type. Since the `SUNMATRIX_CUSPARSE` matrix type is only compatible with the `NVECTOR_CUDA` the restriction is also in place for the linear solver. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

void **SUNLinSol\_cuSolverSp\_batchQR\_GetDescription**(*SUNLinearSolver* LS, char \*\*desc)

The function `SUNLinSol_cuSolverSp_batchQR_GetDescription` accesses the string description of the object (empty by default).

void **SUNLinSol\_cuSolverSp\_batchQR\_SetDescription**(*SUNLinearSolver* LS, const char \*desc)

The function `SUNLinSol_cuSolverSp_batchQR_SetDescription` sets the string description of the object (empty by default).

void **SUNLinSol\_cuSolverSp\_batchQR\_GetDeviceSpace**(*SUNLinearSolver* S, size\_t \*cuSolverInternal, size\_t \*cuSolverWorkspace)

The function `SUNLinSol_cuSolverSp_batchQR_GetDeviceSpace` returns the `cuSOLVER` batch QR method internal buffer size, in bytes, in the argument `cuSolverInternal` and the `cuSOLVER` batch QR workspace buffer size, in bytes, in the argument `cuSolverWorkspace`. The size of the internal buffer is proportional to the number of matrix blocks while the size of the workspace is almost independent of the number of blocks.

### 10.17.3 SUNLinSol\_cuSolverSp\_batchQR content

The `SUNLinSol_cuSolverSp_batchQR` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_cuSolverSp_batchQR {
    int          last_flag;           /* last return flag */
    sunboolean_t first_factorize;     /* is this the first factorization? */
    size_t       internal_size;       /* size of cusolver buffer for Q and R */
    size_t       workspace_size;      /* size of cusolver memory for factorization */
    cusolverSpHandle_t cusolver_handle; /* cuSolverSp context */
    csqrInfo_t    info;               /* opaque cusolver data structure */
    void*         workspace;          /* memory block used by cusolver */
    const char*   desc;               /* description of this linear solver */
};
```

## 10.18 The SUNLINEARSOLVER\_GINKGO Module

Added in version 6.4.0.

The `SUNLINEARSOLVER_GINKGO` implementation of the `SUNLinearSolver` API provides an interface to the linear solvers from the Ginkgo linear algebra library [11]. Since Ginkgo is a modern C++ library, `SUNLINEARSOLVER_GINKGO` is also written in modern C++ (specifically, C++14). Unlike most other SUNDIALS modules, it is a header only library. To use the `SUNLINEARSOLVER_GINKGO` `SUNLinearSolver`, users will need to include `sunlinSol/sunlinSol_ginkgo.hpp`. The module is meant to be used with the `SUNMATRIX_GINKGO` module described in §9.10. Instructions on building SUNDIALS with Ginkgo enabled are given in §17.3.19. For instructions on building and using Ginkgo itself, refer to the [Ginkgo website and documentation](#).

**Note**

It is assumed that users of this module are aware of how to use Ginkgo. This module does not try to encapsulate Ginkgo linear solvers, rather it provides a lightweight interoperability layer between Ginkgo and SUNDIALS. Most, if not all, of the Ginkgo linear solver should work with this interface.

### 10.18.1 Using SUNLINEARSOLVER\_GINKGO

After choosing a compatible `N_Vector` (see §9.10.1) and creating a Ginkgo-enabled `SUNMatrix` (see §9.10) to use the `SUNLINEARSOLVER_GINKGO` module, we first create a Ginkgo stopping criteria object. Importantly, the `sundials::ginkgo::DefaultStop` class provided by SUNDIALS implements a stopping criterion that matches the default SUNDIALS stopping criterion. Namely, it checks if the max iterations (5 by default) were reached or if the absolute residual norm was below a specified tolerance. The criterion can be created just like any other Ginkgo stopping criteria:

```
auto crit{sundials::ginkgo::DefaultStop::build().with_max_iters(max_iters).on(gko_exec)};
```

**Warning**

It is *highly* recommended to employ this criterion when using Ginkgo solvers with SUNDIALS, but it is optional. However, to use the Ginkgo multigrid or cbgmres linear solvers, different Ginkgo criterion must be used.

Once we have created our stopping criterion, we create a Ginkgo solver factory object and wrap it in a `sundials::ginkgo::LinearSolver` object. In this example, we create a Ginkgo conjugate gradient solver:

```
using GkoMatrixType = gko::matrix::Csr<sunrealtype, sunindextype>;
using GkoSolverType = gko::solver::Cg<sunrealtype>;

auto gko_solver_factory = gko::share(
    GkoSolverType::build().with_criteria(std::move(crit)).on(gko_exec));

sundials::ginkgo::LinearSolver<GkoSolverType, GkoMatrixType> LS{
    gko_solver_factory, sunctx};
```

Finally, we can pass the instance of `sundials::ginkgo::LinearSolver` to any function expecting a `SUNLinearSolver` object through the implicit conversion operator or explicit conversion function.

```
// Attach linear solver and matrix to CVODE.
//
// Implicit conversion from sundials::ginkgo::LinearSolver<GkoSolverType, GkoMatrixType>
// to a SUNLinearSolver object is done.
//
// For details about creating A see the SUNMATRIX_GINKGO module.
CvodeSetLinearSolver(cvode_mem, LS, A);

// Alternatively with explicit conversion of LS to a SUNLinearSolver
// and A to a SUNMatrix:
CvodeSetLinearSolver(cvode_mem, LS->get(), A->get());
```

**Warning**

*SUNLinSolFree()* should never be called on a *SUNLinearSolver* that was created via conversion from a *sundials::ginkgo::LinearSolver*. Doing so may result in a double free.

### 10.18.2 SUNLINEARSOLVER\_GINKGO API

In this section we list the public API of the *sundials::ginkgo::LinearSolver* class.

```
template<class GkoSolverType, class GkoMatrixType>
```

```
class sundials::ginkgo::LinearSolver : public sundials::ConvertibleTo<SUNLinearSolver>
```

```
    LinearSolver() = default;
```

Default constructor - means the solver must be moved to.

```
    LinearSolver(std::shared_ptr<typename GkoSolverType::Factory> gko_solver_factory, SUNContext suncctx)
```

Constructs a new *LinearSolver* from a Ginkgo solver factory.

#### Parameters

- **gko\_solver\_factory** – The Ginkgo solver factory (typically *gko::matrix::<type>::Factory*)
- **suncctx** – The SUNDIALS simulation context (*SUNContext*)

```
    LinearSolver(LinearSolver &&that_solver) noexcept
```

Move constructor.

```
    LinearSolver &operator=(LinearSolver &&rhs)
```

Move assignment.

```
    ~LinearSolver() override = default
```

Default destructor.

```
    operator SUNLinearSolver() override
```

Implicit conversion to a *SUNLinearSolver*.

```
    operator SUNLinearSolver() const override
```

Implicit conversion to a *SUNLinearSolver*.

```
    SUNLinearSolver get() override
```

Explicit conversion to a *SUNLinearSolver*.

Added in version 7.6.0: Replaces the *Convert* method which was deprecated.

```
    SUNLinearSolver get() const override
```

Explicit conversion to a *SUNLinearSolver*.

Added in version 7.6.0: Replaces the *Convert* method which was deprecated.

```
    std::shared_ptr<const gko::Executor> GkoExec() const
```

Get the *gko::Executor* associated with the Ginkgo solver.

```
    std::shared_ptr<typename GkoSolverType::Factory> GkoFactory()
```

Get the underlying Ginkgo solver factory.

*GkoSolverType* \***GkoSolver**()

Get the underlying Ginkgo solver.

**Note**

This will be *nullptr* until the linear solver setup phase.

int **NumIters**() const

Get the number of linear solver iterations in the most recent solve.

sunrealtype **ResNorm**() const

Get the residual norm of the solution at the end of the last solve.

The type of residual norm depends on the Ginkgo stopping criteria used with the solver. With the `DefaultStop` criteria this would be the absolute residual 2-norm.

*GkoSolverType* \***Setup**(*Matrix*<*GkoMatrixType*> \*A)

Setup the linear system.

**Parameters**

**A** – the linear system matrix

**Returns**

Pointer to the Ginkgo solver generated from the factory

*gko::LinOp* \***Solve**(*N\_Vector* b, *N\_Vector* x, sunrealtype tol)

Solve the linear system  $Ax = b$  to the specified tolerance.

**Parameters**

- **b** – the right-hand side vector
- **x** – the solution vector
- **tol** – the tolerance to solve the system to

**Returns**

*gko::LinOp*\* the solution

## 10.19 The SUNLINEARSOLVER\_GINKGOBATCH Module

Added in version 7.5.0.

The SUNLINEARSOLVER\_GINKGOBATCH implementation of the `SUNLinearSolver` API provides an interface to the batched linear solvers from the Ginkgo linear algebra library [11]. Like SUNLINEARSOLVER\_GINKGO, this module is written in C++17 and is distributed as a header file. To use the SUNLINEARSOLVER\_GINKGOBATCH `SUNLinearSolver`, users will need to include `sunlinsol/sunlinsol_ginkgobatch.hpp`. The module is meant to be used with the SUNMATRIX\_GINKGOBATCH module described in §9.11. Instructions on building SUNDIALS with Ginkgo enabled are given in §17.3.19. For instructions on building and using Ginkgo itself, refer to the [Ginkgo website and documentation](#).

**Note**

It is assumed that users of this module are aware of how to use Ginkgo. This module does not try to encapsulate Ginkgo linear solvers, rather it provides a lightweight interoperability layer between Ginkgo and SUNDIALS. Most, if not all, of the Ginkgo linear solvers should work with this interface.

### 10.19.1 Using SUNLINEARSOLVER\_GINKGOBATCH

After choosing a compatible `N_Vector` (see §9.11.1) and creating a Ginkgo-enabled `SUNMatrix` (see §9.11) to use the `SUNLINEARSOLVER_GINKGOBATCH` module, we create the linear solver object:

```
using GkoBatchMatrixType = gko::batch::matrix::Csr<sunrealtype, sunindextype>;
using GkoBatchSolverType = gko::batch::solver::Bicgstab<sunrealtype>;
using SUNGkoMatrixType   = sundials::ginkgo::BatchMatrix<GkoBatchMatrixType>;
using SUNGkoLinearSolverType =
    sundials::ginkgo::BatchLinearSolver<GkoBatchSolverType, GkoBatchMatrixType>;

SUNGkoLinearSolverType LS{gko_exec, gko::batch::stop::tolerance_type::absolute,
    precondition_factory, num_batches, sunctx};
```

Next, we can pass the instance of `sundials::ginkgo::BatchLinearSolver` to any function expecting a `SUNLinearSolver` object through the implicit conversion operator or explicit conversion function. For example,

```
// Attach linear solver and matrix to CVODE.
//
// Implicit conversion from sundials::ginkgo::BatchLinearSolver<GkoBatchSolverType, GkoBatchMatrixType>
// to a SUNLinearSolver object is done.
//
// For details about creating A see the SUNMATRIX_GINKGOBATCH module.
CvodeSetLinearSolver(cvode_mem, LS, A);

// Alternatively with explicit conversion of LS to a SUNLinearSolver
// and A to a SUNMatrix:
CvodeSetLinearSolver(cvode_mem, LS.get(), A.get());
```

After attaching the linear solver to the SUNDIALS integrator, one must change the norm factor the integrator uses since the Ginkgo linear solver will take norms over individual batches, not the entire system.

```
// When using ARKODE:
ARKodeSetLSNormFactor(arkode_mem, std::sqrt(batch_size));

// When using CVODE:
CvodeSetLSNormFactor(cvode_mem, std::sqrt(batch_size));

// When using IDA:
IDASetLSNormFactor(ida_mem, std::sqrt(batch_size));
```

#### Warning

Setting the linear solver norm factor is essential. If this is not set, you will likely see a large number of linear solver convergence failures.



**Warning**

`SUNLinSolFree()` should never be called on a `SUNLinearSolver` that was created via conversion from a `sundials::ginkgo::BatchLinearSolver`. Doing so may result in a double free.

**10.19.2 SUNLINEARSOLVER\_GINKGOBATCH API**

All *core functions* `<SUNLinSol.CoreFn>` of the `SUNLinearSolver` API are supported by this module/class. However, we note a difference in behavior for `SUNLinSolNumIters()`:

int **SUNLinSolNumIters\_GinkgoBatch**(*SUNLinearSolver* S)

This function returns the average number of iterations across all of the batch systems. As such, functions that utilize this function to accumulate statistics over steps or solve (i.e., the `GetNumLinIters` function in each package) will return the sum of these average values.

The public API of the `sundials::ginkgo::BatchLinearSolver` class is as follows:

```
template<class GkoBatchSolverType, class GkoBatchMatType>
```

```
class sundials::ginkgo::BatchLinearSolver : public sundials::ConvertibleTo<SUNLinearSolver>
```

```
    BatchLinearSolver(std::shared_ptr<const gko::Executor> gko_exec, sunindextype num_batches,
                      SUNContext sunctx)
```

Constructs a new `BatchLinearSolver` with default tolerance type and max iterations.

**Parameters**

- **gko\_exec** – The `gko::Executor` to use
- **num\_batches** – Number of batches (batch systems)
- **sunctx** – The SUNDIALS simulation context ([\*SUNContext\*](#))

```
    BatchLinearSolver(std::shared_ptr<const gko::Executor> gko_exec, gko::batch::stop::tolerance_type
                      tolerance_type, sunindextype num_batches, SUNContext sunctx)
```

Constructs a new `BatchLinearSolver` with specified tolerance type.

**Parameters**

- **gko\_exec** – The `gko::Executor` to use
- **tolerance\_type** – Ginkgo batch solver tolerance type
- **num\_batches** – Number of batches (batch systems)
- **sunctx** – The SUNDIALS simulation context ([\*SUNContext\*](#))

```
    BatchLinearSolver(std::shared_ptr<const gko::Executor> gko_exec,
                      std::shared_ptr<gko::batch::BatchLinOpFactory> precon_factory, sunindextype
                      num_batches, SUNContext sunctx)
```

Constructs a new `BatchLinearSolver` with a preconditioner factory.

**Parameters**

- **gko\_exec** – The `gko::Executor` to use
- **precon\_factory** – Ginkgo batch preconditioner factory
- **num\_batches** – Number of batches (batch systems)
- **sunctx** – The SUNDIALS simulation context ([\*SUNContext\*](#))

**BatchLinearSolver**(std::shared\_ptr<const gko::Executor> gko\_exec, int max\_iters, sunindextype num\_batches, SUNContext sunctx)

Constructs a new BatchLinearSolver with a maximum number of iterations.

**Parameters**

- **gko\_exec** – The *gko::Executor* to use
- **max\_iters** – Maximum number of iterations
- **num\_batches** – Number of batches (batch systems)
- **sunctx** – The SUNDIALS simulation context (*SUNContext*)

**BatchLinearSolver**(std::shared\_ptr<const gko::Executor> gko\_exec, gko::batch::stop::tolerance\_type tolerance\_type, int max\_iters, sunindextype num\_batches, SUNContext sunctx)

Constructs a new BatchLinearSolver with specified tolerance type and maximum iterations.

**Parameters**

- **gko\_exec** – The *gko::Executor* to use
- **tolerance\_type** – Ginkgo batch solver tolerance type
- **max\_iters** – Maximum number of iterations
- **num\_batches** – Number of batches (batch systems)
- **sunctx** – The SUNDIALS simulation context (*SUNContext*)

**BatchLinearSolver**(std::shared\_ptr<const gko::Executor> gko\_exec,  
std::shared\_ptr<gko::batch::BatchLinOpFactory> precon\_factory, int max\_iters,  
sunindextype num\_batches, SUNContext sunctx)

Constructs a new BatchLinearSolver with a preconditioner factory and maximum iterations.

**Parameters**

- **gko\_exec** – The *gko::Executor* to use
- **precon\_factory** – Ginkgo batch preconditioner factory
- **max\_iters** – Maximum number of iterations
- **num\_batches** – Number of batches (batch systems)
- **sunctx** – The SUNDIALS simulation context (*SUNContext*)

**BatchLinearSolver**(std::shared\_ptr<const gko::Executor> gko\_exec, gko::batch::stop::tolerance\_type tolerance\_type, std::shared\_ptr<gko::batch::BatchLinOpFactory> precon\_factory,  
sunindextype num\_batches, SUNContext sunctx)

Constructs a new BatchLinearSolver with specified tolerance type and preconditioner factory.

**Parameters**

- **gko\_exec** – The *gko::Executor* to use
- **tolerance\_type** – Ginkgo batch solver tolerance type
- **precon\_factory** – Ginkgo batch preconditioner factory
- **num\_batches** – Number of batches (batch systems)
- **sunctx** – The SUNDIALS simulation context (*SUNContext*)

**BatchLinearSolver**(std::shared\_ptr<const gko::Executor> gko\_exec, gko::batch::stop::tolerance\_type tolerance\_type, std::shared\_ptr<gko::batch::BatchLinOpFactory> precon\_factory, int max\_iters, sunindextype num\_batches, SUNContext suncctx)

Constructs a new BatchLinearSolver with all options specified.

#### Parameters

- **gko\_exec** – The *gko::Executor* to use
- **tolerance\_type** – Ginkgo batch solver tolerance type
- **precon\_factory** – Ginkgo batch preconditioner factory
- **max\_iters** – Maximum number of iterations
- **num\_batches** – Number of batches (batch systems)
- **suncctx** – The SUNDIALS simulation context (*SUNContext*)

**BatchLinearSolver**(*BatchLinearSolver* &&that\_solver) noexcept

Move constructor.

*BatchLinearSolver* &**operator=**(*BatchLinearSolver* &&rhs)

Move assignment.

**~BatchLinearSolver**() override = default

Default destructor.

**operator SUNLinearSolver**() override

Implicit conversion to a *SUNLinearSolver*.

**operator SUNLinearSolver**() const override

Implicit conversion to a *SUNLinearSolver*.

*SUNLinearSolver* **get**() override

Explicit conversion to a *SUNLinearSolver*.

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

*SUNLinearSolver* **get**() const override

Explicit conversion to a *SUNLinearSolver*.

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

std::shared\_ptr<const gko::Executor> **GkoExec**() const

Get the *gko::Executor* associated with the Ginkgo solver.

std::shared\_ptr<typename *GkoBatchSolverType*::Factory> **GkoFactory**()

Get the underlying Ginkgo solver factory.

*GkoBatchSolverType* \***GkoSolver**()

Get the underlying Ginkgo solver.

#### Note

This will be nullptr until the linear solver setup phase.

int **AvgNumIters**() const

Get the average number of linear solver iterations across the batches in the most recent solve.

int **StddevNumIters**() const

Get the standard deviation of the number of iterations across the batches during the last solve.

int **SumAvgNumIters**() const

Get the running sum of the average number of iterations in this solver's lifetime.

SUNErrCode **SetScalingMode**(int scaling\_mode)

Sets the matrix scaling mode. The options are:

- `BatchLinearSolver::NO_SCALING` – no scaling of the matrix
- `BatchLinearSolver::LAGGED_SCALING` – the matrix is only scaled when it is updated, this is the default
- `BatchLinearSolver::SOLVE_SCALING` – the matrix is scaled (and unscaled) every solve, this is the most expensive option on a per-solve basis

SUNErrCode **SetScalingVectors**(N\_Vector s1, N\_Vector s2)

Sets the left (s1) and right (s2) scaling vectors to use.

int **Setup**(*BatchMatrix*<*GkoBatchMatType*> \*A)

Setup the linear system.

**Parameters**

**A** – the linear system matrix

**Returns**

Zero on success otherwise a non-zero value

int **Solve**(*BatchMatrix*<*GkoBatchMatType*> \*A, N\_Vector b, N\_Vector x, sunrealtype tol)

Solve the linear system  $Ax = b$  to the specified tolerance.

**Parameters**

- **A** – the linear system matrix
- **b** – the right-hand side vector
- **x** – the solution vector
- **tol** – the tolerance to solve the system to

**Returns**

Zero on success otherwise a non-zero value

## 10.20 The SUNLINEARSOLVER\_KOKKOSDENSE Module

Added in version 6.4.0.

The `SUNLINEARSOLVER_KOKKOSDENSE` [\*SUNLinearSolver\*](#) implementation provides an interface to KokkosKernels [118] linear solvers for dense and batched dense (block-diagonal) systems. Since Kokkos is a modern C++ library, the module is also written in modern C++ (it requires C++14) as a header only library. To utilize this `SUNLinearSolver` user will need to include `sunlinsol/sunlinsol_kokkosdense.hpp`. More instructions on building SUNDIALS with Kokkos and KokkosKernels enabled are given in §17.3.24. For instructions on building and using Kokkos and KokkosKernels, refer to the [Kokkos](#) and [KokkosKernels](#). documentation.

### 10.20.1 Using SUNLINEARSOLVER\_KOKKOSDENSE

The SUNLINEARSOLVER\_KOKKOSDENSE module is defined by the DenseLinearSolver templated class in the sundials::kokkos namespace:

```
template<class ExecSpace = Kokkos::DefaultExecutionSpace,
         class MemSpace = typename ExecSpace::memory_space>
class DenseLinearSolver : public sundials::impl::BaseLinearSolver,
                         public sundials::ConvertibleTo<SUNLinearSolver>
```

To use the SUNLINEARSOLVER\_KOKKOSDENSE module, we begin by constructing an instance of a dense linear solver e.g.,

```
// Create a dense linear solver
sundials::kokkos::DenseLinearSolver<> LS{sunctx};
```

Instances of the DenseLinearSolver class are implicitly or explicitly (using the *get()* method) convertible to a *SUNLinearSolver* e.g.,

```
sundials::kokkos::DenseLinearSolver<> LS{sunctx};
SUNLinearSolver LSA = LS;           // implicit conversion to SUNLinearSolver
SUNLinearSolver LSB = LS.get();    // explicit conversion to SUNLinearSolver
```

#### Warning

*SUNLinSolFree()* should never be called on a *SUNLinearSolver* that was created via conversion from a *sundials::kokkos::DenseLinearSolver*. Doing so may result in a double free.

The SUNLINEARSOLVER\_KOKKOSDENSE module is compatible with the NVECTOR\_KOKKOS vector module (see §8.14) and SUNMATRIX\_KOKKOSDENSE matrix module (see §9.12).

### 10.20.2 SUNLINEARSOLVER\_KOKKOSDENSE API

In this section we list the public API of the *sundials::kokkos::DenseLinearSolver* class.

```
template<class ExecSpace = Kokkos::DefaultExecutionSpace, class MemSpace = typename
ExecSpace::memory_space>
class DenseLinearSolver : public sundials::impl::BaseLinearSolver, public
sundials::ConvertibleTo<SUNLinearSolver>
```

**DenseLinearSolver()** = default;

Default constructor - means the solver must be moved to.

**DenseLinearSolver(SUNContext sunctx)**

Constructs a new DenseLinearSolver.

#### Parameters

**sunctx** – The SUNDIALS simulation context (*SUNContext*)

**DenseLinearSolver(DenseLinearSolver &&that\_solver) noexcept**

Move constructor.

*DenseLinearSolver* &**operator=**(*DenseLinearSolver* &&rhs)

Move assignment.

**~DenseLinearSolver()** override = default

Default destructor.

**operator SUNLinearSolver()** override

Implicit conversion to a [SUNLinearSolver](#).

**operator SUNLinearSolver()** const override

Implicit conversion to a [SUNLinearSolver](#).

**SUNLinearSolver get()** override

Explicit conversion to a [SUNLinearSolver](#).

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

**SUNLinearSolver get()** const override

Explicit conversion to a [SUNLinearSolver](#).

Added in version 7.6.0: Replaces the `Convert` method which was deprecated.

## 10.21 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_SUNLinSolGetID`: Verifies the returned solver identifier against the value that should be returned.
- `Test_SUNLinSolInitialize`: Verifies that `SUNLinSolInitialize` can be called and returns successfully.
- `Test_SUNLinSolSetup`: Verifies that `SUNLinSolSetup` can be called and returns successfully.
- `Test_SUNLinSolSolve`: Given a `SUNMatrix` object  $A$ , `N_Vector` objects  $x$  and  $b$  (where  $Ax = b$ ) and a desired solution tolerance `tol`, this routine clones  $x$  into a new vector  $y$ , calls `SUNLinSolSolve` to fill  $y$  as the solution to  $Ay = b$  (to the input tolerance), verifies that each entry in  $x$  and  $y$  match to within  $10 \times \text{tol}$ , 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_SUNLinSolSetZeroGuess` (iterative solvers only): Verifies that `SUNLinSolSetZeroGuess` can be called and returns successfully.
- `Test_SUNLinSolLastFlag`: Verifies that `SUNLinSolLastFlag` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolNumIters` (iterative solvers only): Verifies that `SUNLinSolNumIters` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolResNorm` (iterative solvers only): Verifies that `SUNLinSolResNorm` can be called, and that the result is non-negative.

- `Test_SUNLinSolResid` (iterative solvers only): Verifies that `SUNLinSolResid` can be called.
- `Test_SUNLinSolSpace` verifies that `SUNLinSolSpace` can be called, and outputs the results to `stdout`.

We'll note that these tests should be performed in a particular order. For either direct or iterative linear solvers, `Test_SUNLinSolInitialize` must be called before `Test_SUNLinSolSetup`, which must be called before `Test_SUNLinSolSolve`. Additionally, for iterative linear solvers `Test_SUNLinSolSetATimes`, `Test_SUNLinSolSetPreconditioner` and `Test_SUNLinSolSetScalingVectors` should be called before `Test_SUNLinSolInitialize`; similarly `Test_SUNLinSolNumIters`, `Test_SUNLinSolResNorm` and `Test_SUNLinSolResid` should be called after `Test_SUNLinSolSolve`. These are called in the appropriate order in all of the example problems.





# Chapter 11

## Nonlinear Algebraic Solvers

SUNDIALS time integration packages are written in terms of generic nonlinear solver operations defined by the SUNNonlinSol API and implemented by a particular SUNNonlinSol module of type `SUNNonlinearSolver`. Users can supply their own SUNNonlinSol module, or use one of the modules provided with SUNDIALS. Depending on the package, nonlinear solver modules can either target systems presented in a rootfinding ( $F(y) = 0$ ) or fixed-point ( $G(y) = y$ ) formulation. For more information on the formulation of the nonlinear system(s) in ARKODE, see §11.2.

The time integrators in SUNDIALS specify a default nonlinear solver module and as such this chapter is intended for users that wish to use a non-default nonlinear solver module or would like to provide their own nonlinear solver implementation. Users interested in using a non-default solver module may skip the description of the SUNNonlinSol API in section §11.1 and proceed to the subsequent sections in this chapter that describe the SUNNonlinSol modules provided with SUNDIALS.

For users interested in providing their own SUNNonlinSol module, the following section presents the SUNNonlinSol API and its implementation beginning with the definition of SUNNonlinSol functions in the sections §11.1.1, §11.1.2 and §11.1.3. This is followed by the definition of functions supplied to a nonlinear solver implementation in the section §11.1.4. The nonlinear solver return codes are given in the section §11.1.5. The `SUNNonlinearSolver` type and the generic SUNNonlinSol module are defined in the section §11.1.6. Finally, the section §11.1.7 lists the requirements for supplying a custom SUNNonlinSol module. Users wishing to supply their own SUNNonlinSol module are encouraged to use the SUNNonlinSol implementations provided with SUNDIALS as templates for supplying custom nonlinear solver modules.

### 11.1 The SUNNonlinearSolver API

The SUNNonlinSol API defines several nonlinear solver operations that enable SUNDIALS integrators to utilize any SUNNonlinSol implementation that provides the required functions. These functions can be divided into three categories. The first are the core nonlinear solver functions. The second consists of “set” routines to supply the nonlinear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of “get” routines for retrieving nonlinear solver statistics. All of these functions are defined in the header file `sundials/sundials_nonlinearsolver.h`.

#### 11.1.1 SUNNonlinearSolver core functions

The core nonlinear solver functions consist of two required functions to get the nonlinear solver type (`SUNNonlinSolGetType()`) and solve the nonlinear system (`SUNNonlinSolSolve()`). The remaining three functions for nonlinear solver initialization (`SUNNonlinSolInitialize()`), setup (`SUNNonlinSolSetup()`), and destruction (`SUNNonlinSolFree()`) are optional.

**enum SUNNonlinearSolver\_Type**

An identifier indicating the form of the nonlinear system expected by the nonlinear solver.

**enumerator SUNNONLINEARSOLVER\_ROOTFIND**

The nonlinear solver expects systems in rootfinding form  $F(y) = 0$

**enumerator SUNNONLINEARSOLVER\_FIXEDPOINT**

The nonlinear solver expects systems in fixed-point form  $G(y) = y$ .

*SUNNonlinearSolver\_Type* **SUNNonlinSolGetType**(*SUNNonlinearSolver* NLS)

This *required* function returns the nonlinear solver type.

**Arguments:**

- *NLS* – a SUNNonlinSol object.

**Return value:**

The *SUNNonlinearSolver\_Type* type identifier for the nonlinear solver.

*SUNErrCode* **SUNNonlinSolInitialize**(*SUNNonlinearSolver* NLS)

This *optional* function handles nonlinear solver initialization and may perform any necessary memory allocations.

**Arguments:**

- *NLS* – a SUNNonlinSol object.

**Return value:**

A *SUNErrCode*.

**Notes:**

It is assumed all solver-specific options have been set prior to calling *SUNNonlinSolInitialize()*. SUNNonlinSol implementations that do not require initialization may set this operation to NULL.

*SUNErrCode* **SUNNonlinSolSetup**(*SUNNonlinearSolver* NLS, *N\_Vector* y, void \*mem)

This *optional* function performs any solver setup needed for a nonlinear solve.

**Arguments:**

- *NLS* – a SUNNonlinSol object.
- *y* – the initial guess passed to the nonlinear solver.
- *mem* – the SUNDIALS integrator memory structure.

**Return value:**

A *SUNErrCode*.

**Notes:**

SUNDIALS integrators call *SUNNonlinSolSetup()* before each step attempt. SUNNonlinSol implementations that do not require setup may set this operation to NULL.

int **SUNNonlinSolSolve**(*SUNNonlinearSolver* NLS, *N\_Vector* y0, *N\_Vector* ycor, *N\_Vector* w, *sunrealtype* tol, *sunbooleantype* callLSetup, void \*mem)

This *required* function solves the nonlinear system  $F(y) = 0$  or  $G(y) = y$ .

**Arguments:**

- *NLS* – a SUNNonlinSol object.
- *y0* – the predicted value for the new solution state. This *must* remain unchanged throughout the solution process.

- *ycor* – on input the initial guess for the correction to the predicted state (zero) and on output the final correction to the predicted state.
- *w* – the solution error weight vector used for computing weighted error norms.
- *tol* – the requested solution tolerance in the weighted root-mean-squared norm.
- *callLSetup* – a flag indicating that the integrator recommends for the linear solver setup function to be called.
- *mem* – the SUNDIALS integrator memory structure.

**Return value:**

The return value is zero for a successful solve, a positive value for a recoverable error (i.e., the solve failed and the integrator should reduce the step size and reattempt the step), and a negative value for an unrecoverable error (i.e., the solve failed and the integrator should halt and return an error to the user).

*SUNErrCode* **SUNNonlinSolFree**(*SUNNonlinearSolver* NLS)

This *optional* function frees any memory allocated by the nonlinear solver.

**Arguments:**

- *NLS* – a *SUNNonlinSol* object.

**Return value:**

- A *SUNErrCode*

### 11.1.2 SUNNonlinearSolver “set” functions

The following functions are used to supply nonlinear solver modules with functions defined by the SUNDIALS integrators and to modify solver parameters. Only the routine for setting the nonlinear system defining function (*SUNNonlinSolSetSysFn()*) is required. All other set functions are optional.

*SUNErrCode* **SUNNonlinSolSetOptions**(*SUNNonlinearSolver* NLS, const char \*NLSid, const char \*file\_name, int argc, char \*argv[])

This *optional* routine sets *SUNNonlinearSolver* options from an array of strings or a file.

**Parameters**

- *NLS* – the *SUNNonlinearSolver* object.
- *NLSid* – the prefix for options to read. The default is “sunnonlinearsolver”.
- *file\_name* – the name of a file containing options to read. If this is NULL or an empty string, “”, then no file is read.
- *argc* – length of the *argv* array.
- *argv* – an array of strings containing the options to set and their values.

**Returns**

*SUNErrCode* indicating success or failure.

**Note**

The *argc* and *argv* arguments are typically those supplied to the user’s *main* routine however, this is not required. The inputs are left unchanged by *SUNNonlinSolSetOptions()*.

If the *NLSid* argument is NULL, then the default prefix, *sunnonlinearsolver*, must be used for all *SUNNonlinearSolver* options. Whether *NLSid* is supplied or not, a “.” must be used to separate an option key

from the prefix. For example, when using the default `NLSid`, the option `sunnonlinearsolver.max_iters` followed by the value can be used to set the maximum number of nonlinear solver iterations. When using a combination of `SUNNonlinearSolver` objects (e.g., when using `MRISetp`), it is recommended that users call `SUNNonlinSolSetOptions()` for each nonlinear solver using distinct `NLSid` inputs, so that each solver object can be configured separately.

`SUNNonlinearSolver` options set via command-line arguments to `SUNNonlinSolSetOptions()` will overwrite any previously-set values. Options are set in the order they are given in `argv` and, if an option with the same prefix appears multiple times in `argv`, the value of the last occurrence will be used.

The supported option names are noted within the documentation for the corresponding “set” function. For options that take a `sunbooltype` as input, use 1 to indicate `true` and 0 for `false`.

### Warning

This function is not available in the Fortran interface.

File-based options are not yet supported, so the `file_name` argument should be set to either `NULL` or the empty string `""`.

Added in version 7.5.0.

**SUNErrCode** `SUNNonlinSolSetSysFn(SUNNonlinearSolver NLS, SUNNonlinSolSysFn SysFn)`

This *required* function is used to provide the nonlinear solver with the function defining the nonlinear system. This is the function  $F(y)$  in  $F(y) = 0$  for `SUNNONLINEARSOLVER_ROOTFIND` modules or  $G(y)$  in  $G(y) = y$  for `SUNNONLINEARSOLVER_FIXEDPOINT` modules.

### Arguments:

- *NLS* – a `SUNNonlinSol` object.
- *SysFn* – the function defining the nonlinear system. See §11.1.4 for the definition of `SUNNonlinSolSysFn`.

### Return value:

- A `SUNErrCode`

**SUNErrCode** `SUNNonlinSolSetLSetupFn(SUNNonlinearSolver NLS, SUNNonlinSolLSetupFn SetupFn)`

This *optional* function is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver setup function.

### Arguments:

- *NLS* – a `SUNNonlinSol` object.
- *SetupFn* – a wrapper function to the SUNDIALS integrator’s linear solver setup function. See §11.1.4 for the definition of `SUNNonlinSolLSetupFn`.

### Return value:

- A `SUNErrCode`

### Notes:

The `SUNNonlinSolLSetupFn` function sets up the linear system  $Ax = b$  where  $A = \frac{\partial F}{\partial y}$  is the linearization of the nonlinear residual function  $F(y) = 0$  (when using `SUNLinSol` direct linear solvers) or calls the user-defined preconditioner setup function (when using `SUNLinSol` iterative linear solvers). `SUNNonlinSol` implementations that do not require solving this system, do not utilize `SUNLinSol` linear solvers, or use `SUNLinSol` linear solvers that do not require setup may set this operation to `NULL`.

*SUNErrCode* **SUNNonlinSolSetLSolveFn**(*SUNNonlinearSolver* NLS, *SUNNonlinSolLSolveFn* SolveFn)

This *optional* function is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver solve function.

**Arguments:**

- *NLS* – a *SUNNonlinSol* object.
- *SolveFn* – a wrapper function to the SUNDIALS integrator’s linear solver solve function. See §11.1.4 for the definition of *SUNNonlinSolLSolveFn*.

**Return value:**

- A *SUNErrCode*

**Notes:**

The *SUNNonlinSolLSolveFn* function solves the linear system  $Ax = b$  where  $A = \frac{\partial F}{\partial y}$  is the linearization of the nonlinear residual function  $F(y) = 0$ . *SUNNonlinSol* implementations that do not require solving this system or do not use *SUNLinSol* linear solvers may set this operation to *NULL*.

*SUNErrCode* **SUNNonlinSolSetConvTestFn**(*SUNNonlinearSolver* NLS, *SUNNonlinSolConvTestFn* CTestFn, void \*ctest\_data)

This *optional* function is used to provide the nonlinear solver with a function for determining if the nonlinear solver iteration has converged. This is typically called by SUNDIALS integrators to define their nonlinear convergence criteria, but may be replaced by the user.

**Arguments:**

- *NLS* – a *SUNNonlinSol* object.
- *CTestFn* – a SUNDIALS integrator’s nonlinear solver convergence test function. See §11.1.4 for the definition of *SUNNonlinSolConvTestFn*.
- *ctest\_data* – is a data pointer passed to *CTestFn* every time it is called.

**Return value:**

- A *SUNErrCode*

**Notes:**

*SUNNonlinSol* implementations utilizing their own convergence test criteria may set this function to *NULL*.

*SUNErrCode* **SUNNonlinSolSetMaxIters**(*SUNNonlinearSolver* NLS, int maxiters)

This *optional* function sets the maximum number of nonlinear solver iterations. This is typically called by SUNDIALS integrators to define their default iteration limit, but may be adjusted by the user.

**Arguments:**

- *NLS* – a *SUNNonlinSol* object.
- *maxiters* – the maximum number of nonlinear iterations.

**Return value:**

- A *SUNErrCode*

**Notes:**

If supported by the *SUNNonlinearSolver* implementation, this routine will be called by *SUNNonlinSolSetOptions()* when using the key “NLSid.max\_iters”.

### 11.1.3 SUNNonlinearSolver “get” functions

The following functions allow SUNDIALS integrators to retrieve nonlinear solver statistics. The routines to get the number of iterations in the most recent solve (*SUNNonlinSolGetNumIters()*) and number of convergence failures are optional. The routine to get the current nonlinear solver iteration (*SUNNonlinSolGetCurIter()*) is required when using the convergence test provided by the SUNDIALS integrator or when using an iterative SUNLinSol linear solver module; otherwise *SUNNonlinSolGetCurIter()* is optional.

*SUNErrCode* **SUNNonlinSolGetNumIters**(*SUNNonlinearSolver* NLS, long int \*nitters)

This *optional* function returns the number of nonlinear solver iterations in the most recent solve. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.

**Arguments:**

- *NLS* – a SUNNonlinSol object.
- *nitters* – the total number of nonlinear solver iterations.

**Return value:**

- A *SUNErrCode*

*SUNErrCode* **SUNNonlinSolGetCurIter**(*SUNNonlinearSolver* NLS, int \*iter)

This function returns the iteration index of the current nonlinear solve. This function is *required* when using SUNDIALS integrator-provided convergence tests or when using an iterative SUNLinSol linear solver module; otherwise it is *optional*.

**Arguments:**

- *NLS* – a SUNNonlinSol object.
- *iter* – the nonlinear solver iteration in the current solve starting from zero.

**Return value:**

- A *SUNErrCode*

*SUNErrCode* **SUNNonlinSolGetNumConvFails**(*SUNNonlinearSolver* NLS, long int \*nconvfails)

This *optional* function returns the number of nonlinear solver convergence failures in the most recent solve. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.

**Arguments:**

- *NLS* – a SUNNonlinSol object.
- *nconvfails* – the total number of nonlinear solver convergence failures.

**Return value:**

- A *SUNErrCode*

### 11.1.4 Functions provided by SUNDIALS integrators

To interface with SUNNonlinSol modules, the SUNDIALS integrators supply a variety of routines for evaluating the nonlinear system, calling the SUNLinSol setup and solve functions, and testing the nonlinear iteration for convergence. These integrator-provided routines translate between the user-supplied ODE or DAE systems and the generic interfaces to the nonlinear or linear systems of equations that result in their solution. The functions provided to a SUNNonlinSol module have types defined in the header file `sundials/sundials_nonlinearsolver.h`; these are also described below.

```
typedef int (*SUNNonlinSolSysFn)(N_Vector ycor, N_Vector F, void *mem)
```

These functions evaluate the nonlinear system  $F(y)$  for `SUNNONLINEARSOLVER_ROOTFIND` type modules or  $G(y)$  for `SUNNONLINEARSOLVER_FIXEDPOINT` type modules. Memory for  $F$  must be allocated prior to calling this function. The vector  $ycor$  will be left unchanged.

**Arguments:**

- $ycor$  – is the current correction to the predicted state at which the nonlinear system should be evaluated.
- $F$  – is the output vector containing  $F(y)$  or  $G(y)$ , depending on the solver type.
- $mem$  – is the SUNDIALS integrator memory structure.

**Return value:**

The return value is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

**Notes:**

SUNDIALS integrators formulate nonlinear systems as a function of the correction to the predicted solution. On each call to the nonlinear system function the integrator will compute and store the current solution based on the input correction. Additionally, the residual will store the value of the ODE right-hand side function or DAE residual used in computing the nonlinear system. These stored values are then directly used in the integrator-supplied linear solver setup and solve functions as applicable.

```
typedef int (*SUNNonlinSolLSetupFn)(sunbooleantype jbad, sunbooleantype *jcur, void *mem)
```

These functions are wrappers to the SUNDIALS integrator's function for setting up linear solves with `SUNLinSol` modules.

**Arguments:**

- $jbad$  – is an input indicating whether the nonlinear solver believes that  $A$  has gone stale (`SUNTRUE`) or not (`SUNFALSE`).
- $jcur$  – is an output indicating whether the routine has updated the Jacobian  $A$  (`SUNTRUE`) or not (`SUNFALSE`).
- $mem$  – is the SUNDIALS integrator memory structure.

**Return value:**

The return value is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

**Notes:**

The `SUNNonlinSolLSetupFn` function sets up the linear system  $Ax = b$  where  $A = \frac{\partial F}{\partial y}$  is the linearization of the nonlinear residual function  $F(y) = 0$  (when using `SUNLinSol` direct linear solvers) or calls the user-defined preconditioner setup function (when using `SUNLinSol` iterative linear solvers). `SUNNonlinSol` implementations that do not require solving this system, do not utilize `SUNLinSol` linear solvers, or use `SUNLinSol` linear solvers that do not require setup may ignore these functions.

As discussed in the description of `SUNNonlinSolSysFn`, the linear solver setup function assumes that the nonlinear system function has been called prior to the linear solver setup function as the setup will utilize saved values from the nonlinear system evaluation (e.g., the updated solution).

```
typedef int (*SUNNonlinSolLSolveFn)(N_Vector b, void *mem)
```

These functions are wrappers to the SUNDIALS integrator's function for solving linear systems with `SUNLinSol` modules.

**Arguments:**

- $b$  – contains the right-hand side vector for the linear solve on input and the solution to the linear system on output.

- *mem* – is the SUNDIALS integrator memory structure.

**Return value:**

The return value is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

**Notes:**

The *SUNNonlinSolLSolveFn* function solves the linear system  $Ax = b$  where  $A = \frac{\partial F}{\partial y}$  is the linearization of the nonlinear residual function  $F(y) = 0$ . SUNNonlinSol implementations that do not require solving this system or do not use SUNLinSol linear solvers may ignore these functions.

As discussed in the description of *SUNNonlinSolSysFn*, the linear solver solve function assumes that the nonlinear system function has been called prior to the linear solver solve function as the setup may utilize saved values from the nonlinear system evaluation (e.g., the updated solution).

```
typedef int (*SUNNonlinSolConvTestFn)(SUNNonlinearSolver NLS, N_Vector ycor, N_Vector del, sunrealtype tol, N_Vector ewt, void *ctest_data)
```

These functions are SUNDIALS integrator-specific convergence tests for nonlinear solvers and are typically supplied by each SUNDIALS integrator, but users may supply custom problem-specific versions as desired.

**Arguments:**

- *NLS* – is the SUNNonlinSol object.
- *ycor* – is the current correction (nonlinear iterate).
- *del* – is the difference between the current and prior nonlinear iterates.
- *tol* – is the nonlinear solver tolerance.
- *ewt* – is the weight vector used in computing weighted norms.
- *ctest\_data* – is the data pointer provided to *SUNNonlinSolSetConvTestFn()*.

**Return value:**

The return value of this routine will be a negative value if an unrecoverable error occurred or one of the following:

- SUN\_SUCCESS – the iteration is converged.
- SUN\_NLS\_CONTINUE – the iteration has not converged, keep iterating.
- SUN\_NLS\_CONV\_RECVR – the iteration appears to be diverging, try to recover.

**Notes:**

The tolerance passed to this routine by SUNDIALS integrators is the tolerance in a weighted root-mean-squared norm with error weight vector *ewt*. SUNNonlinSol modules utilizing their own convergence criteria may ignore these functions.

### 11.1.5 SUNNonlinearSolver return codes

The functions provided to SUNNonlinSol modules by each SUNDIALS integrator, and functions within the SUNDIALS-provided SUNNonlinSol implementations, utilize a common set of return codes shown in [Table 11.1](#). Here, negative values correspond to non-recoverable failures, positive values to recoverable failures, and zero to a successful call.



Table 11.1: Description of the SUNNonlinearSolver return codes.

Name	Value	Description
SUN_SUCCESS	0	successful call or converged solve
SUN_NLS_CONTINUE	901	the nonlinear solver is not converged, keep iterating
SUN_NLS_CONV_RECVR	902	the nonlinear solver appears to be diverging, try to recover

### 11.1.6 The generic SUNNonlinearSolver module

SUNDIALS integrators interact with specific SUNNonlinSol implementations through the generic SUNNonlinSol module on which all other SUNNonlinSol implementations are built. The SUNNonlinearSolver type is a pointer to a structure containing an implementation-dependent *content* field and an *ops* field.

A *SUNNonlinearSolver* is a pointer to the *\_generic\_SUNNonlinearSolver* structure:

```
typedef struct _generic_SUNNonlinearSolver *SUNNonlinearSolver
```

```
struct _generic_SUNNonlinearSolver
```

The structure defining the SUNDIALS nonlinear solver class.

void **\*content**

Pointer to nonlinear solver-specific member data

*SUNNonlinearSolver\_Ops* **ops**

A virtual table of nonlinear solver operations provided by a specific implementation

*SUNContext* **sunctx**

The SUNDIALS simulation context

The virtual table structure is defined as

```
typedef struct _generic_SUNNonlinearSolver_Ops *SUNNonlinearSolver_Ops
```

```
struct _generic_SUNNonlinearSolver_Ops
```

The structure defining *SUNNonlinearSolver* operations.

*SUNNonlinearSolver\_Type* (**\*gettype**)(*SUNNonlinearSolver*)

The function implementing *SUNNonlinSolGetType()*

int (**\*initialize**)(*SUNNonlinearSolver*)

The function implementing *SUNNonlinSolInitialize()*

int (**\*setup**)(*SUNNonlinearSolver*, *N\_Vector*, void\*)

The function implementing *SUNNonlinSolSetup()*

int (**\*solve**)(*SUNNonlinearSolver*, *N\_Vector*, *N\_Vector*, *N\_Vector*, *sunrealtype*, *sunbooleantype*, void\*)

The function implementing *SUNNonlinSolSolve()*

int (**\*free**)(*SUNNonlinearSolver*)

The function implementing *SUNNonlinSolFree()*

int (**\*setsysfn**)(*SUNNonlinearSolver*, *SUNNonlinSolSysFn*)

The function implementing *SUNNonlinSolSetSysFn()*

int (**\*setlsetupfn**)(*SUNNonlinearSolver*, *SUNNonlinSolLSetupFn*)

The function implementing *SUNNonlinSolSetLSetupFn()*

```
int (*setlsolvefn)(SUNNonlinearSolver, SUNNonlinSolLSolveFn)
```

The function implementing `SUNNonlinSolSetLSolveFn()`

```
int (*setctestfn)(SUNNonlinearSolver, SUNNonlinSolConvTestFn, void*)
```

The function implementing `SUNNonlinSolSetConvTestFn()`

```
int (*setmaxiters)(SUNNonlinearSolver, int)
```

The function implementing `SUNNonlinSolSetMaxIters()`

```
int (*getnumiters)(SUNNonlinearSolver, long int*)
```

The function implementing `SUNNonlinSolGetNumIters()`

```
int (*getcuriter)(SUNNonlinearSolver, int*)
```

The function implementing `SUNNonlinSolGetCurIter()`

```
int (*getnumconvfails)(SUNNonlinearSolver, long int*)
```

The function implementing `SUNNonlinSolGetNumConvFails()`

The generic `SUNNonlinSol` module defines and implements the nonlinear solver operations defined in §11.1.1–§11.1.3. These routines are in fact only wrappers to the nonlinear solver operations provided by a particular `SUNNonlinSol` implementation, which are accessed through the `ops` field of the `SUNNonlinearSolver` structure. To illustrate this point we show below the implementation of a typical nonlinear solver operation from the generic `SUNNonlinSol` module, namely `SUNNonlinSolSolve()`, which solves the nonlinear system and returns a flag denoting a successful or failed solve:

```
int SUNNonlinSolSolve(SUNNonlinearSolver NLS,
                     N_Vector y0, N_Vector y,
                     N_Vector w, sunrealtype tol,
                     sunbooleantype callLSetup, void* mem)
{
    return((int) NLS->ops->solve(NLS, y0, y, w, tol, callLSetup, mem));
}
```

### 11.1.7 Implementing a Custom SUNNonlinearSolver Module

A `SUNNonlinSol` implementation *must* do the following:

- Specify the content of the `SUNNonlinSol` module.
- Define and implement the required nonlinear solver operations defined in §11.1.1–§11.1.3. Note that the names of the module routines should be unique to that implementation in order to permit using more than one `SUNNonlinSol` module (each with different `SUNNonlinearSolver` internal data representations) in the same code.
- Define and implement a user-callable constructor to create a `SUNNonlinearSolver` object.
- If the implementation depends on a linear solver, then it must evaluate the nonlinear system function passed to `SUNNonlinSolSetSysFn()` *before* setting up the linear solver (signaled by the `callLSetup` flag to `SUNNonlinSolSolve()`).

To aid in the creation of custom `SUNNonlinearSolver` modules, the generic `SUNNonlinearSolver` module provides the utility functions `SUNNonlinSolNewEmpty()` and `SUNNonlinSolFreeEmpty()`. When used in custom `SUNNonlinearSolver` constructors these functions will ease the introduction of any new optional nonlinear solver operations to the `SUNNonlinearSolver` API by ensuring that only required operations need to be set.

`SUNNonlinearSolver` **SUNNonlinSolNewEmpty**(`SUNContext` sunctx)

This function allocates a new generic `SUNNonlinearSolver` object and initializes its content pointer and the function pointers in the operations structure to NULL.

**Return value:**

If successful, this function returns a `SUNNonlinearSolver` object. If an error occurs when allocating the object, then this routine will return `NULL`.

void **SUNNonlinSolFreeEmpty**(*SUNNonlinearSolver* NLS)

This routine frees the generic `SUNNonlinearSolver` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is `NULL`, and, if it is not, it will free it as well.

**Arguments:**

- *NLS* – a `SUNNonlinearSolver` object

Additionally, a `SUNNonlinearSolver` implementation *may* do the following:

- Define and implement additional user-callable “set” routines acting on the `SUNNonlinearSolver` object, e.g., for setting various configuration options to tune the performance of the nonlinear solve algorithm.
- Provide additional user-callable “get” routines acting on the `SUNNonlinearSolver` object, e.g., for returning various solve statistics.

## 11.2 ARKODE SUNNonlinearSolver interface

As discussed in §2 integration steps often require the (approximate) solution of nonlinear systems. These systems can be formulated as the rootfinding problem

$$\begin{aligned} G(z_i) &\equiv z_i - \gamma f^I(t_{n,i}^I, z_i) - a_i = 0 & [M = I], \\ G(z_i) &\equiv M z_i - \gamma f^I(t_{n,i}^I, z_i) - a_i = 0 & [M \text{ static}], \\ G(z_i) &\equiv M(t_{n,i}^I)(z_i - a_i) - \gamma f^I(t_{n,i}^I, z_i) = 0 & [M \text{ time-dependent}], \end{aligned}$$

where  $z_i$  is the  $i$ -th stage at time  $t_i$  and  $a_i$  is known data that depends on the integration method.

Alternately, the nonlinear system above may be formulated as the fixed-point problem

$$z_i = z_i - M(t_{n,i}^I)^{-1} G(z_i),$$

where  $G(z_i)$  is the variant of the rootfinding problem listed above, and  $M(t_{n,i}^I)$  may equal either  $M$  or  $I$ , as applicable.

Rather than solving the above nonlinear systems for the stage value  $z_i$  directly, ARKODE modules solve for the correction  $z_{cor}$  to the predicted stage value  $z_{pred}$  so that  $z_i = z_{pred} + z_{cor}$ . Thus these nonlinear systems rewritten in terms of  $z_{cor}$  are

$$\begin{aligned} G(z_{cor}) &\equiv z_{cor} - \gamma f^I(t_{n,i}^I, z_i) - \tilde{a}_i = 0 & [M = I], \\ G(z_{cor}) &\equiv M z_{cor} - \gamma f^I(t_{n,i}^I, z_i) - \tilde{a}_i = 0 & [M \text{ static}], \\ G(z_{cor}) &\equiv M(t_{n,i}^I)(z_{cor} - \tilde{a}_i) - \gamma f^I(t_{n,i}^I, z_i) = 0 & [M \text{ time-dependent}], \end{aligned} \tag{11.1}$$

for the rootfinding problem and

$$z_{cor} = z_{cor} - M(t_{n,i}^I)^{-1} G(z_i), \tag{11.2}$$

for the fixed-point problem.

The nonlinear system functions provided by ARKODE modules to the nonlinear solver module internally update the current value of the stage based on the input correction vector i.e.,  $z_i = z_{pred} + z_{cor}$ . The updated vector  $z_i$  is used when calling the ODE right-hand side function and when setting up linear solves (e.g., updating the Jacobian or preconditioner).

ARKODE modules also provide several advanced functions that will not be needed by most users, but might be useful for users who choose to provide their own `SUNNonlinSol` implementation for use by ARKODE. These routines provide access to the internal integrator data required to evaluate (11.1) or (11.2).

### 11.2.1 ARKODE advanced output functions

Two notable functions were already listed in §5.3.10.1:

- `ARKodeGetCurrentState()` – returns the current state vector. When called within the computation of a step (i.e., during a nonlinear solve) this is the current stage state vector  $z_i = z_{pred} + z_{cor}$ . Otherwise this is the current internal solution state vector  $y(t)$ . In either case the corresponding stage or solution time can be obtained from `ARKodeGetCurrentTime()`.
- `ARKodeGetCurrentGamma()` – returns the current value of the scalar  $\gamma$ .

Additional advanced output functions that are provided to aid in the construction of user-supplied SUNNonlinSol modules are as follows.

int **ARKodeGetCurrentMassMatrix**(void \*arkode\_mem, *SUNMatrix* \*M)

Returns the current mass matrix. For a time dependent mass matrix the corresponding time can be obtained from `ARKodeGetCurrentTime()`.

**Arguments:**

- *arkode\_mem* – pointer to the ARKODE memory block.
- *M* – *SUNMatrix* pointer that will get set to the current mass matrix  $M(t)$ . If a matrix-free method is used the output is NULL.

**Return value:**

- ARK\_SUCCESS if successful.
- ARK\_MEM\_NULL if the ARKStep memory was NULL.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

int **ARKodeGetNonlinearSystemData**(void \*arkode\_mem, *sunrealtype* \*tcur, *N\_Vector* \*zpred, *N\_Vector* \*z, *N\_Vector* \*Fi, *sunrealtype* \*gamma, *N\_Vector* \*sdata, void \*\*user\_data)

Returns all internal data required to construct the current nonlinear implicit system (11.1) or (11.2):

**Arguments:**

- *arkode\_mem* – pointer to the ARKODE memory block.
- *tcur* – value of the independent variable corresponding to implicit stage,  $t_{n,i}^I$ .
- *zpred* – the predicted stage vector  $z_{pred}$  at  $t_{n,i}^I$ . This vector must not be changed.
- *z* – the stage vector  $z_i$  above. This vector may be not current and may need to be filled (see the note below).
- *Fi* – the implicit function evaluated at the current time and state,  $f^I(t_{n,i}^I, z_i)$ . This vector may be not current and may need to be filled (see the note below).
- *gamma* – current  $\gamma$  for implicit stage calculation.
- *sdata* – accumulated data from previous solution and stages,  $\tilde{a}_i$ . This vector must not be changed.
- *user\_data* – pointer to the user-defined data structure (as specified through `ARKodeSetUserData()`, or NULL otherwise)

**Return value:**

- ARK\_SUCCESS if successful.

- ARK\_MEM\_NULL if the ARKODE memory was NULL.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

This routine is intended for users who wish to attach a custom `SUNNonlinSolSysFn` to an existing `SUNNonlinearSolver` object (through a call to `SUNNonlinSolSetSysFn()`) or who need access to nonlinear system data to compute the nonlinear system function as part of a custom `SUNNonlinearSolver` object.

When supplying a custom `SUNNonlinSolSysFn` to an existing `SUNNonlinearSolver` object, the user should call `ARKodeGetNonlinearSystemData()` **inside** the nonlinear system function to access the requisite data for evaluating the nonlinear system function of their choosing. Additionally, if the `SUNNonlinearSolver` object (existing or custom) leverages the `SUNNonlinSolSetupFn` and/or `SUNNonlinSolSolveFn` functions supplied by ARKODE (through calls to `SUNNonlinSolSetLSetupFn()` and `SUNNonlinSolSetLSolveFn()` respectively) the vectors  $z$  and  $Fi$  **must be filled in** by the user's `SUNNonlinSolSysFn` with the current state and corresponding evaluation of the right-hand side function respectively i.e.,

$$\begin{aligned} z &= z_{pred} + z_{cor}, \\ Fi &= f^I(t_{n,i}^I, z_i), \end{aligned}$$

where  $z_{cor}$  was the first argument supplied to the `SUNNonlinSolSysFn`.

If this function is called as part of a custom linear solver (i.e., the default `SUNNonlinSolSysFn` is used) then the vectors  $z$  and  $Fi$  are only current when `ARKodeGetNonlinearSystemData()` is called after an evaluation of the nonlinear system function.

int **ARKodeComputeState**(void \*arkode\_mem, *N\_Vector* zcor, *N\_Vector* z)

Computes the current stage state vector using the stored prediction and the supplied correction from the nonlinear solver i.e.,  $z_i(t) = z_{pred} + z_{cor}$ .

**Arguments:**

- *arkode\_mem* – pointer to the ARKODE memory block.
- *zcor* – the correction from the nonlinear solver.
- *z* – on output, the current stage state vector  $z_i$ .

**Return value:**

- ARK\_SUCCESS if successful.
- ARK\_MEM\_NULL if the ARKODE memory was NULL.

**Note**

This is only compatible with time-stepping modules that support implicit algebraic solvers.

## 11.2.2 ARKStep advanced output functions (deprecated)

Two notable functions were already listed in §5.7.1.10:

- `ARKStepGetCurrentState()` – returns the current state vector. When called within the computation of a step (i.e., during a nonlinear solve) this is the current stage state vector  $z_i = z_{pred} + z_{cor}$ . Otherwise this is the current

internal solution state vector  $y(t)$ . In either case the corresponding stage or solution time can be obtained from [ARKStepGetCurrentTime\(\)](#).

- [ARKStepGetCurrentGamma\(\)](#) – returns the current value of the scalar  $\gamma$ .

Additional advanced output functions that are provided to aid in the construction of user-supplied SUNNonlinSol modules are as follows.

int **ARKStepGetCurrentMassMatrix**(void \*arkode\_mem, [SUNMatrix](#) \*M)

Returns the current mass matrix. For a time dependent mass matrix the corresponding time can be obtained from [ARKStepGetCurrentTime\(\)](#).

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *M* – SUNMatrix pointer that will get set to the current mass matrix  $M(t)$ . If a matrix-free method is used the output is NULL.

**Return value:**

- ARK\_SUCCESS if successful.
- ARK\_MEM\_NULL if the ARKStep memory was NULL.

Deprecated since version 6.1.0: Use [ARKodeGetCurrentMassMatrix\(\)](#) instead.

int **ARKStepGetNonlinearSystemData**(void \*arkode\_mem, [sunrealtype](#) \*tcur, [N\\_Vector](#) \*zpred, [N\\_Vector](#) \*z, [N\\_Vector](#) \*Fi, [sunrealtype](#) \*gamma, [N\\_Vector](#) \*sdata, void \*\*user\_data)

Returns all internal data required to construct the current nonlinear implicit system (11.1) or (11.2):

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *tcur* – value of the independent variable corresponding to implicit stage,  $t_{n,i}^I$ .
- *zpred* – the predicted stage vector  $z_{pred}$  at  $t_{n,i}^I$ . This vector must not be changed.
- *z* – the stage vector  $z_i$  above. This vector may be not current and may need to be filled (see the note below).
- *Fi* – the implicit function evaluated at the current time and state,  $f^I(t_{n,i}^I, z_i)$ . This vector may be not current and may need to be filled (see the note below).
- *gamma* – current  $\gamma$  for implicit stage calculation.
- *sdata* – accumulated data from previous solution and stages,  $\tilde{a}_i$ . This vector must not be changed.
- *user\_data* – pointer to the user-defined data structure (as specified through [ARKStepSetUserData\(\)](#), or NULL otherwise)

**Return value:**

- ARK\_SUCCESS if successful.
- ARK\_MEM\_NULL if the ARKStep memory was NULL.

**Note**

This routine is intended for users who wish to attach a custom [SUNNonlinSolSysFn](#) to an existing SUN-NonlinearSolver object (through a call to [SUNNonlinSolSetSysFn\(\)](#)) or who need access to nonlinear system data to compute the nonlinear system function as part of a custom SUNNonlinearSolver object.

When supplying a custom `SUNNonlinSolSysFn` to an existing `SUNNonlinearSolver` object, the user should call `ARKStepGetNonlinearSystemData()` **inside** the nonlinear system function to access the requisite data for evaluating the nonlinear system function of their choosing. Additionally, if the `SUNNonlinearSolver` object (existing or custom) leverages the `SUNNonlinSolLSetupFn` and/or `SUNNonlinSolLSolveFn` functions supplied by ARKStep (through calls to `SUNNonlinSolSetLSetupFn()` and `SUNNonlinSolSetLSolveFn()` respectively) the vectors  $z$  and  $Fi$  **must be filled in** by the user's `SUNNonlinSolSysFn` with the current state and corresponding evaluation of the right-hand side function respectively i.e.,

$$z = z_{pred} + z_{cor},$$

$$Fi = f^I(t_{n,i}^I, z_i),$$

where  $z_{cor}$  was the first argument supplied to the `SUNNonlinSolSysFn`.

If this function is called as part of a custom linear solver (i.e., the default `SUNNonlinSolSysFn` is used) then the vectors  $z$  and  $Fi$  are only current when `ARKStepGetNonlinearSystemData()` is called after an evaluation of the nonlinear system function.

Deprecated since version 6.1.0: Use `ARKodeGetNonlinearSystemData()` instead.

int `ARKStepComputeState`(void \*arkode\_mem, *N\_Vector* zcor, *N\_Vector* z)

Computes the current stage state vector using the stored prediction and the supplied correction from the nonlinear solver i.e.,  $z_i(t) = z_{pred} + z_{cor}$ .

**Arguments:**

- *arkode\_mem* – pointer to the ARKStep memory block.
- *zcor* – the correction from the nonlinear solver.
- *z* – on output, the current stage state vector  $z_i$ .

**Return value:**

- `ARK_SUCCESS` if successful.
- `ARK_MEM_NULL` if the ARKStep memory was NULL.

Deprecated since version 6.1.0: Use `ARKodeComputeState()` instead.

### 11.2.3 MRISStep advanced output functions (deprecated)

Two notable functions were already listed in §5.11.2.9:

- `MRISStepGetCurrentState()` – returns the current state vector. When called within the computation of a step (i.e., during a nonlinear solve) this is the current stage state vector  $z_i = z_{pred} + z_{cor}$ . Otherwise this is the current internal solution state vector  $y(t)$ . In either case the corresponding stage or solution time can be obtained from `MRISStepGetCurrentTime()`.
- `MRISStepGetCurrentGamma()` – returns the current value of the scalar  $\gamma$ .

Additional advanced output functions that are provided to aid in the construction of user-supplied `SUNNonlinSol` modules are as follows.

int `MRISStepGetNonlinearSystemData`(void \*arkode\_mem, *sunrealtype* \*tcur, *N\_Vector* \*zpred, *N\_Vector* \*z, *N\_Vector* \*Fi, *sunrealtype* \*gamma, *N\_Vector* \*sdata, void \*\*user\_data)

Returns all internal data required to construct the current nonlinear implicit system (11.1) or (11.2):

**Arguments:**



- *arkode\_mem* – pointer to the MRISStep memory block.
- *tcur* – value of independent variable corresponding to slow stage ( $t_{n,i}^S$  above).
- *zpred* – predicted nonlinear solution ( $z_{pred}$  above). This vector must not be changed.
- *z* – stage vector ( $z_i$  above). This vector may be not current and may need to be filled (see the note below).
- *Fi* – memory available for evaluating the slow implicit RHS ( $f^I(t_{n,i}^S, z_i)$  above). This vector may be not current and may need to be filled (see the note below).
- *gamma* – current  $\gamma$  for slow stage calculation.
- *sdata* – accumulated data from previous solution and stages ( $\tilde{a}_i$  above). This vector must not be changed.
- *user\_data* – pointer to the user-defined data structure (as specified through *MRISStepSetUserData()*, or NULL otherwise).

**Return value:**

- ARK\_SUCCESS if successful.
- ARK\_MEM\_NULL if the MRISStep memory was NULL.

**Note**

This routine is intended for users who wish to attach a custom *SUNNonlinSolSysFn* to an existing *SUNNonlinearSolver* object (through a call to *SUNNonlinSolSetSysFn()*) or who need access to nonlinear system data to compute the nonlinear system function as part of a custom *SUNNonlinearSolver* object.

When supplying a custom *SUNNonlinSolSysFn* to an existing *SUNNonlinearSolver* object, the user should call *MRISStepGetNonlinearSystemData()* **inside** the nonlinear system function to access the requisite data for evaluating the nonlinear system function of their choosing. Additionally, if the *SUNNonlinearSolver* object (existing or custom) leverages the *SUNNonlinSolLSetupFn* and/or *SUNNonlinSolLSolveFn* functions supplied by MRISStep (through calls to *SUNNonlinSolSetLSetupFn()* and *SUNNonlinSolSetLSolveFn()* respectively) the vectors *z* and *F* **must be filled** in by the user's *SUNNonlinSolSysFn* with the current state and corresponding evaluation of the right-hand side function respectively i.e.,

$$\begin{aligned} z &= z_{pred} + z_{cor}, \\ Fi &= f^I(t_{n,i}^S, z_i), \end{aligned}$$

where  $z_{cor}$  was the first argument supplied to the *SUNNonlinSolSysFn*.

If this function is called as part of a custom linear solver (i.e., the default *SUNNonlinSolSysFn* is used) then the vectors *z* and *Fi* are only current when *MRISStepGetNonlinearSystemData()* is called after an evaluation of the nonlinear system function.

Deprecated since version 6.1.0: Use *ARKodeGetNonlinearSystemData()* instead.

int **MRISStepComputeState**(void \*arkode\_mem, *N\_Vector* zcor, *N\_Vector* z)

Computes the current stage state vector using the stored prediction and the supplied correction from the nonlinear solver i.e.,  $z_i = z_{pred} + z_{cor}$ .

**Arguments:**

- *arkode\_mem* – pointer to the MRISStep memory block.
- *zcor* – the correction from the nonlinear solver.



- $z$  – on output, the current stage state vector  $z_i$ .

**Return value:**

- ARK\_SUCCESS if successful.
- ARK\_MEM\_NULL if the MRISStep memory was NULL.

Deprecated since version 6.1.0: Use [ARKodeComputeState\(\)](#) instead.

## 11.3 The SUNNonlinSol\_Newton implementation

This section describes the SUNNonlinSol implementation of Newton's method. To access the SUNNonlinSol\_Newton module, include the header file `sunnonlinSol/sunnonlinSol_newton.h`. We note that the SUNNonlinSol\_Newton module is accessible from SUNDIALS integrators *without* separately linking to the `libsundials_sunnonlinSol_newton` module library.

### 11.3.1 SUNNonlinSol\_Newton description

To find the solution to

$$F(y) = 0 \tag{11.3}$$

given an initial guess  $y^{(0)}$ , Newton's method computes a series of approximate solutions

$$y^{(m+1)} = y^{(m)} + \delta^{(m+1)}$$

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

$$A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}), \tag{11.4}$$

in which  $A$  is the Jacobian matrix

$$A \equiv \partial F / \partial y. \tag{11.5}$$

Depending on the linear solver used, the SUNNonlinSol\_Newton module will employ either a Modified Newton method or an Inexact Newton method [21, 24, 33, 35, 67]. When used with a direct linear solver, the Jacobian matrix  $A$  is held constant during the Newton iteration, resulting in a Modified Newton method. With a matrix-free iterative linear solver, the iteration is an Inexact Newton method.

In both cases, calls to the integrator-supplied [SUNNonlinSolSetupFn](#) function are made infrequently to amortize the increased cost of matrix operations (updating  $A$  and its factorization within direct linear solvers, or updating the preconditioner within iterative linear solvers). Specifically, SUNNonlinSol\_Newton will call the [SUNNonlinSolSetupFn](#) function in two instances:

- when requested by the integrator (the input `callSetSetup` is `SUNTRUE`) before attempting the Newton iteration, or
- when reattempting the nonlinear solve after a recoverable failure occurs in the Newton iteration with stale Jacobian information (`jcur` is `SUNFALSE`). In this case, SUNNonlinSol\_Newton will set `jbad` to `SUNTRUE` before calling the [SUNNonlinSolSetupFn\(\)](#) function.

Whether the Jacobian matrix  $A$  is fully or partially updated depends on logic unique to each integrator-supplied [SUNNonlinSolSetupFn](#) routine. We refer to the discussion of nonlinear solver strategies provided in the package-specific Mathematics section of the documentation for details.

The default maximum number of iterations and the stopping criteria for the Newton iteration are supplied by the SUNDIALS integrator when SUNNonlinSol\_Newton is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling the [SUNNonlinSolSetMaxIters\(\)](#) and/or [SUNNonlinSolSetConvTestFn\(\)](#) functions after attaching the SUNNonlinSol\_Newton object to the integrator.

### 11.3.2 SUNNonlinSol\_Newton functions

The SUNNonlinSol\_Newton module provides the following constructor for creating the SUNNonlinearSolver object.

*SUNNonlinearSolver* **SUNNonlinSol\_Newton**(*N\_Vector* y, *SUNContext* sunctx)

This creates a SUNNonlinearSolver object for use with SUNDIALS integrators to solve nonlinear systems of the form  $F(y) = 0$  using Newton's method.

**Arguments:**

- y – a template for cloning vectors needed within the solver.
- sunctx – the *SUNContext* object (see §4.2)

**Return value:**

A SUNNonlinSol object if the constructor exits successfully, otherwise it will be NULL.

The SUNNonlinSol\_Newton module implements all of the functions defined in §11.1.1–§11.1.3 except for *SUNNonlinSolSetup()*. The SUNNonlinSol\_Newton functions have the same names as those defined by the generic SUNNonlinSol API with *\_Newton* appended to the function name. Unless using the SUNNonlinSol\_Newton module as a standalone nonlinear solver the generic functions defined in §11.1.1–§11.1.3 should be called in favor of the SUNNonlinSol\_Newton-specific implementations.

The SUNNonlinSol\_Newton module also defines the following user-callable function.

*SUNErrCode* **SUNNonlinSolGetSysFn\_Newton**(*SUNNonlinearSolver* NLS, *SUNNonlinSolSysFn* \*SysFn)

This returns the residual function that defines the nonlinear system.

**Arguments:**

- NLS – a SUNNonlinSol object.
- SysFn – the function defining the nonlinear system.

**Return value:**

- A *SUNErrCode*

**Notes:**

This function is intended for users that wish to evaluate the nonlinear residual in a custom convergence test function for the SUNNonlinSol\_Newton module. We note that SUNNonlinSol\_Newton will not leverage the results from any user calls to *SysFn*.

### 11.3.3 SUNNonlinSol\_Newton content

The *content* field of the SUNNonlinSol\_Newton module is the following structure.

```
struct _SUNNonlinearSolverContent_Newton {

    SUNNonlinSolSysFn      Sys;
    SUNNonlinSolLSetupFn   LSetup;
    SUNNonlinSolLSolveFn   LSolve;
    SUNNonlinSolConvTestFn CTest;

    N_Vector               delta;
    sunbooleantype          jcur;
    int                     curiter;
    int                     maxiters;
```

(continues on next page)

(continued from previous page)

```

long int      niters;
long int      nconvfails;
void*         ctest_data;
};

```

These entries of the *content* field contain the following information:

- **Sys** – the function for evaluating the nonlinear system,
- **LSetup** – the package-supplied function for setting up the linear solver,
- **LSolve** – the package-supplied function for performing a linear solve,
- **CTest** – the function for checking convergence of the Newton iteration,
- **delta** – the Newton iteration update vector,
- **jcur** – the Jacobian status (SUNTRUE = current, SUNFALSE = stale),
- **curiter** – the current number of iterations in the solve attempt,
- **maxiters** – the maximum number of Newton iterations allowed in a solve,
- **niters** – the total number of nonlinear iterations across all solves,
- **nconvfails** – the total number of nonlinear convergence failures across all solves,
- **ctest\_data** – the data pointer passed to the convergence test function,

## 11.4 The SUNNonlinSol\_FixedPoint implementation

This section describes the SUNNonlinSol implementation of a fixed point (functional) iteration with optional Anderson acceleration. To access the SUNNonlinSol\_FixedPoint module, include the header file `sunnonlinSol/sunnonlinSol_fixedpoint.h`. We note that the SUNNonlinSol\_FixedPoint module is accessible from SUNDIALS integrators *without* separately linking to the `libsundials_sunnonlinSolfixedpoint` module library.

### 11.4.1 SUNNonlinSol\_FixedPoint description

To find the solution to

$$G(y) = y \quad (11.6)$$

given an initial guess  $y^{(0)}$ , the fixed point iteration computes a series of approximate solutions

$$y^{(n+1)} = G(y^{(n)}) \quad (11.7)$$

where  $n$  is the iteration index. The convergence of this iteration may be accelerated using Anderson's method [10, 42, 79, 125]. With Anderson acceleration using subspace size  $m$ , the series of approximate solutions can be formulated as the linear combination

$$y^{(n+1)} = \beta \sum_{i=0}^{m_n} \alpha_i^{(n)} G(y^{(n-m_n+i)}) + (1 - \beta) \sum_{i=0}^{m_n} \alpha_i^{(n)} y_{n-m_n+i} \quad (11.8)$$

where  $m_n = \min \{m, n\}$  and the factors

$$\alpha^{(n)} = (\alpha_0^{(n)}, \dots, \alpha_{m_n}^{(n)})$$

solve the minimization problem  $\min_{\alpha} \|F_n \alpha^T\|_2$  under the constraint that  $\sum_{i=0}^{m_n} \alpha_i = 1$  where

$$F_n = (f_{n-m_n}, \dots, f_n)$$

with  $f_i = G(y^{(i)}) - y^{(i)}$ . Due to this constraint, in the limit of  $m = 0$  the accelerated fixed point iteration formula (11.8) simplifies to the standard fixed point iteration (11.7).

Following the recommendations made in [125], the `SUNNonlinSol_FixedPoint` implementation computes the series of approximate solutions as

$$y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} - (1-\beta)(f(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta f_{n-m_n+i}) \quad (11.9)$$

with  $\Delta g_i = G(y^{(i+1)}) - G(y^{(i)})$  and where the factors

$$\gamma^{(n)} = (\gamma_0^{(n)}, \dots, \gamma_{m_n-1}^{(n)})$$

solve the unconstrained minimization problem  $\min_{\gamma} \|f_n - \Delta F_n \gamma^T\|_2$  where

$$\Delta F_n = (\Delta f_{n-m_n}, \dots, \Delta f_{n-1}),$$

with  $\Delta f_i = f_{i+1} - f_i$ . The least-squares problem is solved by applying a QR factorization to  $\Delta F_n = Q_n R_n$  and solving  $R_n \gamma = Q_n^T f_n$ .

The acceleration subspace size  $m$  is required when constructing the `SUNNonlinSol_FixedPoint` object. The default maximum number of iterations and the stopping criteria for the fixed point iteration are supplied by the `SUNDIALS` integrator when `SUNNonlinSol_FixedPoint` is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling `SUNNonlinSolSetMaxIters()` and `SUNNonlinSolSetConvTestFn()` after attaching the `SUNNonlinSol_FixedPoint` object to the integrator.

## 11.4.2 SUNNonlinSol\_FixedPoint functions

The `SUNNonlinSol_FixedPoint` module provides the following constructor for creating the `SUNNonlinearSolver` object.

*SUNNonlinearSolver* **SUNNonlinSol\_FixedPoint**(*N\_Vector* y, int m, *SUNContext* sunctx)

This creates a `SUNNonlinearSolver` object for use with `SUNDIALS` integrators to solve nonlinear systems of the form  $G(y) = y$ .

### Arguments:

- *y* – a template for cloning vectors needed within the solver.
- *m* – the number of acceleration vectors to use.
- *sunctx* – the `SUNContext` object (see §4.2)

### Return value:

A `SUNNonlinSol` object if the constructor exits successfully, otherwise it will be `NULL`.

Since the accelerated fixed point iteration (11.7) does not require the setup or solution of any linear systems, the `SUNNonlinSol_FixedPoint` module implements all of the functions defined in §11.1.1–§11.1.3 except for the `SUNNonlinSolSetup()`, `SUNNonlinSolSetLSetupFn()`, and `SUNNonlinSolSetLSolveFn()` functions, that are set to `NULL`. The `SUNNonlinSol_FixedPoint` functions have the same names as those defined by the generic `SUNNonlinSol` API with `_FixedPoint` appended to the function name. Unless using the `SUNNonlinSol_FixedPoint` module as a standalone nonlinear solver the generic functions defined in §11.1.1–§11.1.3 should be called in favor of the `SUNNonlinSol_FixedPoint`-specific implementations.

The `SUNNonlinSol_FixedPoint` module also defines the following user-callable functions.

*SUNErrCode* **SUNNonlinSolGetSysFn\_FixedPoint**(*SUNNonlinearSolver* NLS, *SUNNonlinSolSysFn* \*SysFn)

This returns the fixed-point function that defines the nonlinear system.

**Arguments:**

- *NLS* – a *SUNNonlinSol* object.
- *SysFn* – the function defining the nonlinear system.

**Return value:**

- A *SUNErrCode*

**Notes:**

This function is intended for users that wish to evaluate the fixed-point function in a custom convergence test function for the *SUNNonlinSol\_FixedPoint* module. We note that *SUNNonlinSol\_FixedPoint* will not leverage the results from any user calls to *SysFn*.

*SUNErrCode* **SUNNonlinSolSetDamping\_FixedPoint**(*SUNNonlinearSolver* NLS, *sunrealtype* beta)

This sets the damping parameter  $\beta$  to use with Anderson acceleration. By default damping is disabled i.e.,  $\beta = 1.0$ .

**Arguments:**

- *NLS* – a *SUNNonlinSol* object.
- *beta* – the damping parameter  $0 < \beta \leq 1$ .

**Return value:**

- A *SUNErrCode*

**Notes:**

A beta value should satisfy  $0 < \beta < 1$  if damping is to be used. A value of one or more will disable damping.

### 11.4.3 SUNNonlinSol\_FixedPoint content

The *content* field of the *SUNNonlinSol\_FixedPoint* module is the following structure.

```
struct _SUNNonlinearSolverContent_FixedPoint {

    SUNNonlinSolSysFn      Sys;
    SUNNonlinSolConvTestFn CTest;

    int                    m;
    int                    *imap;
    sunrealtype            *R;
    sunbooleanype          damping
    sunrealtype            beta
    sunrealtype            *gamma;
    sunrealtype            *cvals;
    N_Vector               *df;
    N_Vector               *dg;
    N_Vector               *q;
    N_Vector               *Xvecs;
    N_Vector               yprev;
    N_Vector               gy;
    N_Vector               fold;
```

(continues on next page)

(continued from previous page)

```
N_Vector      gold;
N_Vector      delta;
int           curiter;
int           maxiters;
long int      niters;
long int      nconvfails;
void          *ctest_data;
};
```

The following entries of the *content* field are always allocated:

- Sys – function for evaluating the nonlinear system,
- CTest – function for checking convergence of the fixed point iteration,
- yprev – N\_Vector used to store previous fixed-point iterate,
- gy – N\_Vector used to store  $G(y)$  in fixed-point algorithm,
- delta – N\_Vector used to store difference between successive fixed-point iterates,
- curiter – the current number of iterations in the solve attempt,
- maxiters – the maximum number of fixed-point iterations allowed in a solve,
- niters – the total number of nonlinear iterations across all solves,
- nconvfails – the total number of nonlinear convergence failures across all solves,
- ctest\_data – the data pointer passed to the convergence test function,
- m – number of acceleration vectors,

If Anderson acceleration is requested (i.e.,  $m > 0$  in the call to [SUNNonlinSol\\_FixedPoint\(\)](#)), then the following items are also allocated within the *content* field:

- imap – index array used in acceleration algorithm (length m),
- damping – a flag indicating if damping is enabled,
- beta – the damping parameter,
- R – small matrix used in acceleration algorithm (length m\*m),
- gamma – small vector used in acceleration algorithm (length m),
- cvals – small vector used in acceleration algorithm (length m+1),
- df – array of vectors used in acceleration algorithm (length m),
- dg – array of vectors used in acceleration algorithm (length m),
- q – array of vectors used in acceleration algorithm (length m),
- Xvecs – vector pointer array used in acceleration algorithm (length m+1),
- fold – vector used in acceleration algorithm, and
- gold – vector used in acceleration algorithm.

## 11.5 The SUNNonlinSol\_PetscSNES implementation

This section describes the SUNNonlinSol interface to the [PETSc SNES nonlinear solver\(s\)](#). To enable the SUNNonlinSol\_PetscSNES module, SUNDIALS must be configured to use PETSc. Instructions on how to do this are given in §17.3.31. To access the SUNNonlinSol\_PetscSNES module, include the header file `sunnonlinSol/sunnonlinSol_petscsnes.h`. The library to link to is `libsundials_sunnonlinSolpetsc.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries. Users of the SUNNonlinSol\_PetscSNES module should also see §8.9 which discusses the NVECTOR interface to the PETSc Vec API.

### 11.5.1 SUNNonlinSol\_PetscSNES description

The SUNNonlinSol\_PetscSNES implementation allows users to utilize a PETSc SNES nonlinear solver to solve the nonlinear systems that arise in the SUNDIALS integrators. Since SNES uses the KSP linear solver interface underneath it, the SUNNonlinSol\_PetscSNES implementation does not interface with SUNDIALS linear solvers. Instead, users should set nonlinear solver options, linear solver options, and preconditioner options through the PETSc SNES, KSP, and PC APIs.

*Important usage notes for the SUNNonlinSol\_PetscSNES implementation:*

- The SUNNonlinSol\_PetscSNES implementation handles calling `SNESSetFunction` at construction. The actual residual function  $F(y)$  is set by the SUNDIALS integrator when the SUNNonlinSol\_PetscSNES object is attached to it. Therefore, a user should not call `SNESSetFunction` on a SNES object that is being used with SUNNonlinSol\_PetscSNES. For these reasons it is recommended, although not always necessary, that the user calls `SUNNonlinSol_PetscSNES()` with the new SNES object immediately after calling `SNESCreate`.
- The number of nonlinear iterations is tracked by SUNDIALS separately from the count kept by SNES. As such, the function `SUNNonlinSolGetNumIters()` reports the cumulative number of iterations across the lifetime of the `SUNNonlinearSolver` object.
- Some “converged” and “diverged” convergence reasons returned by SNES are treated as recoverable convergence failures by SUNDIALS. Therefore, the count of convergence failures returned by `SUNNonlinSolGetNumConvervFails()` will reflect the number of recoverable convergence failures as determined by SUNDIALS, and may differ from the count returned by `SNESGetNonlinearStepFailures`.
- The SUNNonlinSol\_PetscSNES module is not currently compatible with the CVODES or IDAS staggered or simultaneous sensitivity strategies.

### 11.5.2 SUNNonlinearSolver\_PetscSNES functions

The SUNNonlinSol\_PetscSNES module provides the following constructor for creating a `SUNNonlinearSolver` object.

`SUNNonlinearSolver` **SUNNonlinSol\_PetscSNES**(*N\_Vector* y, SNES snes, *SUNContext* sunctx)

This creates a SUNNonlinSol object that wraps a PETSc SNES object for use with SUNDIALS. This will call `SNESSetFunction` on the provided SNES object.

**Arguments:**

- *snes* – a PETSc SNES object.
- *y* – a *N\_Vector* object of type `NVECTOR_PETSC` that is used as a template for the residual vector.
- *sunctx* – the `SUNContext` object (see §4.2)

**Return value:**

A SUNNonlinSol object if the constructor exits successfully, otherwise it will be NULL.

**Warning**

This function calls `SNESSetFunction` and will overwrite whatever function was previously set. Users should not call `SNESSetFunction` on the SNES object provided to the constructor.

The `SUNNonlinSol_PetscSNES` module implements all of the functions defined in §11.1.1–§11.1.3 except for `SUNNonlinSolSetup()`, `SUNNonlinSolSetLSetupFn()`, `SUNNonlinSolSetLSolveFn()`, `SUNNonlinSolSetConvTestFn()`, and `SUNNonlinSolSetMaxIters()`.

The `SUNNonlinSol_PetscSNES` functions have the same names as those defined by the generic `SUNNonlinSol` API with `_PetscSNES` appended to the function name. Unless using the `SUNNonlinSol_PetscSNES` module as a standalone nonlinear solver the generic functions defined in §11.1.1–§11.1.3 should be called in favor of the `SUNNonlinSol_PetscSNES` specific implementations.

The `SUNNonlinSol_PetscSNES` module also defines the following user-callable functions.

*SUNErrCode* **SUNNonlinSolGetSNES\_PetscSNES**(*SUNNonlinearSolver* NLS, SNES \*snes)

This gets the SNES object that was wrapped.

**Arguments:**

- *NLS* – a `SUNNonlinSol` object.
- *snes* – a pointer to a PETSc SNES object that will be set upon return.

**Return value:**

A *SUNErrCode*

*SUNErrCode* **SUNNonlinSolGetPetscError\_PetscSNES**(*SUNNonlinearSolver* NLS, PetscErrorCode \*error)

This gets the last error code returned by the last internal call to a PETSc API function.

**Arguments:**

- *NLS* – a `SUNNonlinSol` object.
- *error* – a pointer to a PETSc error integer that will be set upon return.

**Return value:**

A *SUNErrCode*

*SUNErrCode* **SUNNonlinSolGetSysFn\_PetscSNES**(*SUNNonlinearSolver* NLS, *SUNNonlinSolSysFn* \*SysFn)

This returns the residual function that defines the nonlinear system.

**Arguments:**

- *NLS* – a `SUNNonlinSol` object.
- *SysFn* – the function defining the nonlinear system.

**Return value:**

A *SUNErrCode*

### 11.5.3 SUNNonlinearSolver\_PetscSNES content

The *content* field of the `SUNNonlinSol_PetscSNES` module is the following structure.

```
struct _SUNNonlinearSolverContent_PetscSNES {
    int sysfn_last_err;
    PetscErrorCode petsc_last_err;
```

(continues on next page)



(continued from previous page)

```
long int nconvfails;
long int nni;
void *imem;
SNES snes;
Vec r;
N_Vector y, f;
SUNNonlinSolSysFn Sys;
};
```

These entries of the *content* field contain the following information:

- `sysfn_last_err` – last error returned by the system defining function,
- `petsc_last_err` – last error returned by PETSc,
- `nconvfails` – number of nonlinear converge failures (recoverable or not),
- `nni` – number of nonlinear iterations,
- `imem` – SUNDIALS integrator memory,
- `snes` – PETSc SNES object,
- `r` – the nonlinear residual,
- `y` – wrapper for PETSc vectors used in the system function,
- `f` – wrapper for PETSc vectors used in the system function,
- `Sys` – nonlinear system defining function.



## Chapter 12

# Dominant Eigenvalue Estimators

### 12.1 Introduction to Dominant Eigenvalue Estimators

For problems that require the dominant eigenvalue of a matrix (i.e., the Jacobian), the SUNDIALS packages operate using generic dominant eigenvalue estimator modules defined through the *SUNDomEigEstimator* class. This allows SUNDIALS packages to utilize any valid *SUNDomEigEstimator* implementation that provides a set of required functions. These functions can be divided into three categories. The first are the core estimator functions. The second group consists of “set” routines to supply the dominant eigenvalue estimator object with functions provided by the SUNDIALS package, or for modification of estimator parameters. The last group consists of “get” routines for retrieving artifacts (statistics, residual, etc.) from the estimator. All of these functions are defined in the header file `sundials/sundials_domeigestimator.h`.

The implementations provided with SUNDIALS work in coordination with the SUNDIALS *N\_Vector* modules to provide a set of compatible data structures for the estimator. Moreover, advanced users can provide a customized *SUNDomEigEstimator* implementation to any SUNDIALS package, particularly in cases where they provide their own *N\_Vector*.

While Krylov-based estimators preset the number of Krylov subspace dimensions, resulting in a tolerance-free estimation, SUNDIALS requires that iterative estimators stop when the residual meets a prescribed tolerance,  $\tau$ ,

$$\frac{|\lambda_k - \lambda_{k-1}|}{|\lambda_k|} < \tau. \quad (12.1)$$

For users interested in providing their own *SUNDomEigEstimator()*, the following section presents the *SUNDomEigEstimator* class and its implementation beginning with the definition of *SUNDomEigEstimator* functions in §12.2.1 – §12.2.3. This is followed by the definition of functions supplied to an estimator implementation in §12.2.4. The *SUNDomEigEstimator* type is defined §12.2.5. The section that then follows describes the *SUNDomEigEstimator* functions required by this SUNDIALS package, and provides additional package specific details. Then the remaining sections of this chapter present the *SUNDomEigEstimator* modules provided with SUNDIALS.

### 12.2 The SUNDomEigEstimator API

Added in version 7.5.0.

The *SUNDomEigEst* API defines several dominant eigenvalue estimation operations that enable SUNDIALS packages to utilize this API. These functions can be divided into three categories. The first are the core dominant eigenvalue estimation functions. The second consist of “set” routines to supply the dominant eigenvalue estimator with functions provided by the SUNDIALS packages and to modify estimator parameters. The final group consists of “get” routines

for retrieving dominant eigenvalue estimation statistics. All of these functions are defined in the header file `sundials/sundials_domeigestimator.h`.

### 12.2.1 SUNDomEigEstimator core functions

The `SUNDomEigEstimator` base class provides two **utility** routines for implementers, `SUNDomEigEstimator_NewEmpty()` and `SUNDomEigEstimator_FreeEmpty()`.

Implementations of `SUNDomEigEstimator`s must include a **required** `SUNDomEigEstimator_Estimate()` function to estimate the dominant eigenvalue.

*SUNDomEigEstimator* **SUNDomEigEstimator\_NewEmpty**(*SUNContext* sunctx)

This function allocates a new `SUNDomEigEstimator` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

**Arguments:**

- *sunctx* – the *SUNContext* object (see §4.2).

**Return value:**

If successful, this function returns a `SUNDomEigEstimator` object. If an error occurs when allocating the object, then this routine will return `NULL`.

*SUNErrCode* **SUNDomEigEstimator\_Initialize**(*SUNDomEigEstimator* DEE)

This *optional* function performs dominant eigenvalue estimator initialization (assuming that all estimator-specific options have been set).

**Arguments:**

- *DEE* – a `SUNDomEigEstimator` object.

**Return value:**

A *SUNErrCode*.

*SUNErrCode* **SUNDomEigEstimator\_Estimate**(*SUNDomEigEstimator* DEE, *sunrealtype* \*lambdaR, *sunrealtype* \*lambdaI)

This *required* function estimates the dominant eigenvalue,  $\lambda_{\max} = \lambda_R + \lambda_I i$  such that  $|\lambda| = \max\{|\lambda_i| : A\vec{v}_i = \lambda_i \vec{v}_i, \vec{v}_i \neq \vec{0}\}$ .

**Arguments:**

- *DEE* – a `SUNDomEigEstimator` object.
- *lambdaR* – The real part of the dominant eigenvalue.
- *lambdaI* – The imaginary part of the dominant eigenvalue.

**Return value:**

*SUN\_SUCCESS* for a successful call, or a relevant error code from *SUNErrCode* upon failure.

#### Note

When the estimator is used in a time-dependent context, an implementation may reuse the same initial guess as the initial call to `SUNDomEigEstimator_Estimate()` or use an improved guess based on the result of the most recent `SUNDomEigEstimator_Estimate()` call. See the documentation of the specific *SUNDomEigEstimator* implementation for more details.

*SUNErrCode* **SUNDomEigEstimator\_FreeEmpty**(*SUNDomEigEstimator* DEE)

This routine frees the *SUNDomEigEstimator* object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will free it as well.

**Arguments:**

- *DEE* – a *SUNDomEigEstimator* object.

**Return value:**

A *SUNErrCode*.

*SUNErrCode* **SUNDomEigEstimator\_Destroy**(*SUNDomEigEstimator* \*DEEptr)

Frees memory allocated by the dominant eigenvalue estimator.

**Arguments:**

- *DEEptr* – a *SUNDomEigEstimator* object pointer.

**Return value:**

A *SUNErrCode*.

## 12.2.2 SUNDomEigEstimator “set” functions

The following functions supply dominant eigenvalue estimator modules with functions defined by the SUNDIALS packages and modify estimator parameters. When using the matrix-vector product routine provided by a SUNDIALS integration, the *SetATimes* is required. Otherwise, all set functions are optional. *SUNDomEigEst* implementations that do not provide the functionality for any optional routine should leave the corresponding function pointer NULL instead of supplying a dummy routine.

*SUNErrCode* **SUNDomEigEstimator\_SetOptions**(*SUNDomEigEstimator* DEE, const char \*Did, const char \*file\_name, int argc, char \*argv[])

Sets *SUNDomEigEstimator* options from an array of strings or a file.

**Parameters**

- *DEE* – the *SUNDomEigEstimator* object.
- *Did* – the prefix for options to read. The default is “sundomeigestimator”.
- *file\_name* – the name of a file containing options to read. If this is NULL or an empty string, “”, then no file is read.
- *argc* – number of command-line arguments passed to executable.
- *argv* – an array of strings containing the options to set and their values.

**Returns**

*SUNErrCode* indicating success or failure.

**Note**

The *argc* and *argv* arguments are typically those supplied to the user’s *main* routine however, this is not required. The inputs are left unchanged by *SUNDomEigEstimator\_SetOptions()*.

If the *Did* argument is NULL then the default prefix, *sundomeigestimator*, must be used for all *SUNDomEigEstimator* options. Whether *Did* is supplied or not, a “.” must be used to separate an option key

from the prefix. For example, when using the default `Did`, the option `sundomeigestimator.max_iters` followed by the value can be used to set the maximum number of iterations.

`SUNDomEigEstimator` options set via `SUNDomEigEstimator_SetOptions()` will overwrite any previously-set values. Options are set in the order they are given in `argv` and, if an option with the same prefix appears multiple times in `argv`, the value of the last occurrence will be used.

The supported option names are noted within the documentation for the corresponding `SUNDomEigEstimator` functions. For options that take a *sunboolean* type as input, use 1 to indicate `true` and 0 for `false`.

### Warning

This function is not available in the Fortran interface.

File-based options are not yet supported, so the `file_name` argument should be set to either `NULL` or the empty string `""`.

Added in version 7.5.0.

*SUNErrCode* **SUNDomEigEstimator\_SetATimes**(*SUNDomEigEstimator* DEE, void \*A\_data, *SUNATimesFn* ATimes)

This function provides a *SUNATimesFn* function for performing matrix-vector products, as well as a `void*` pointer to a data structure used by this routine, to the dominant eigenvalue estimator. This function is *required* when using the matrix-vector product function provided by a SUNDIALS integrator, otherwise the function is *optional*.

### Arguments:

- *DEE* – a `SUNDomEigEstimator` object.
- *A\_data* – pointer to structure for `ATimes`.
- *ATimes* – function pointer to perform  $Av$  product.

### Return value:

A *SUNErrCode*.

*SUNErrCode* **SUNDomEigEstimator\_SetNumPreprocessIters**(*SUNDomEigEstimator* DEE, int num\_iters)

This *optional* routine sets the number of preprocessing matrix-vector multiplications, performed at the beginning of each `SUNDomEigEstimator_Estimate()` evaluation.

Applying preprocessing iterations may be useful if the initial guess used in `SUNDomEigEstimator_Estimate()` is not a good approximation of the dominant eigenvector and can help reduce some computational overhead.

### Arguments:

- *DEE* – a `SUNDomEigEstimator` object.
- *num\_iters* – the number of preprocessing iterations. Supplying a value  $< 0$ , will reset the value to the implementation default.

### Return value:

A *SUNErrCode*.

**Note**

When the estimator is used in a time-dependent context, different numbers of preprocessing iterations may be desired for the initial estimate than on subsequent estimations. Thus, when the estimator is used with `LSRKStep` (see `LSRKStepSetDomEigEstimator()`), the initial value of `num_iters` should be set with `LSRKStepSetNumDomEigEstInitPreprocessIters()` while the number of preprocessing iterations for subsequent estimates should be set with `LSRKStepSetNumDomEigEstPreprocessIters()`.

This routine will be called by `SUNDomEigEstimator_SetOptions()` when using the key “Did.num\_preprocess\_iters”.

**SUNErrCode** `SUNDomEigEstimator_SetRelTol`(*SUNDomEigEstimator* DEE, *sunrealtype* rel\_tol)

This *optional* routine sets the estimator’s *relative tolerance*.

**Arguments:**

- *DEE* – a `SUNDomEigEstimator` object.
- *rel\_tol* – the requested eigenvalue accuracy.

**Return value:**

A *SUNErrCode*.

**Note**

This routine will be called by `SUNDomEigEstimator_SetOptions()` when using the key “Did.rel\_tol”.

**SUNErrCode** `SUNDomEigEstimator_SetMaxIters`(*SUNDomEigEstimator* DEE, long int max\_iters)

This *optional* routine sets the maximum number of iterations.

**Arguments:**

- *DEE* – a `SUNDomEigEstimator` object.
- *max\_iters* – the maximum number of iterations.

**Return value:**

A *SUNErrCode*.

**Note**

This routine will be called by `SUNDomEigEstimator_SetOptions()` when using the key “Did.max\_iters”.

**SUNErrCode** `SUNDomEigEstimator_SetInitialGuess`(*SUNDomEigEstimator* DEE, *N\_Vector* q)

This *optional* routine sets the initial vector guess to start with.

The vector *q* does not need to be normalized before this set routine.

**Arguments:**

- *DEE* – a `SUNDomEigEstimator` object.
- *q* – the initial guess vector.

**Return value:**

A *SUNErrCode*.

### 12.2.3 SUNDomEigEstimator “get” functions

The following functions allow SUNDIALS packages to retrieve results from a dominant eigenvalue estimator. *All routines are optional.*

*SUNErrCode* **SUNDomEigEstimator\_GetRes**(*SUNDomEigEstimator* DEE, *sunrealtype* \*cur\_res)

This *optional* routine should return the final residual from the most-recent call to *SUNDomEigEstimator\_Estimate()*.

**Arguments:**

- *DEE* – a *SUNDomEigEstimator* object.
- *cur\_res* – the residual.

**Return value:**

A *SUNErrCode*.

**Usage:**

```
sunrealtype cur_res;  
retval = SUNDomEigEstimator_GetRes(DEE, &cur_res);
```

*SUNErrCode* **SUNDomEigEstimator\_GetNumIters**(*SUNDomEigEstimator* DEE, long int \*num\_iters)

This *optional* routine should return the number of estimator iterations performed in the most-recent call to *SUNDomEigEstimator\_Estimate()*.

**Arguments:**

- *DEE* – a *SUNDomEigEstimator* object.
- *num\_iters* – the number of iterations.

**Return value:**

A *SUNErrCode*.

**Usage:**

```
long int num_iters;  
retval = SUNDomEigEstimator_GetNumIters(DEE, &num_iters);
```

*SUNErrCode* **SUNDomEigEstimator\_GetNumATimesCalls**(*SUNDomEigEstimator* DEE, long int \*num\_ATimes)

This *optional* routine should return the number of calls to the *SUNATimesFn* function.

**Arguments:**

- *DEE* – a *SUNDomEigEstimator* object.
- *num\_ATimes* – the number of calls to the *Atimes* function.

**Return value:**

A *SUNErrCode*.

**Usage:**

```
long int num_ATimes;  
retval = SUNDomEigEstimator_GetNumATimesCalls(DEE, &num_ATimes);
```



*SUNErrCode* **SUNDomEigEstimator\_Write**(*SUNDomEigEstimator* DEE, FILE \*outfile)

This *optional* routine prints the dominant eigenvalue estimator settings to the file pointer.

**Arguments:**

- *DEE* – a *SUNDomEigEstimator* object.
- *outfile* – the output stream.

**Return value:**

A *SUNErrCode*.

## 12.2.4 Functions provided by SUNDIALS packages

To interface with *SUNDomEigEst* modules, the SUNDIALS packages supply a *SUNATimesFn* function for evaluating the matrix-vector product. This package-provided routine translates between the user-supplied ODE or DAE systems and the generic dominant eigenvalue estimator API. The function types for these routines are defined in the header file *sundials/sundials\_iterative.h*.

## 12.2.5 The generic *SUNDomEigEstimator* module

SUNDIALS packages interact with dominant eigenvalue estimator implementations through the *SUNDomEigEstimator* class. A *SUNDomEigEstimator* is a pointer to the *SUNDomEigEstimator\_* structure:

```
typedef struct SUNDomEigEstimator_ *SUNDomEigEstimator
```

```
struct SUNDomEigEstimator_
```

The structure defining the SUNDIALS dominant eigenvalue estimator class.

```
void *content
```

Pointer to the dominant eigenvalue estimator-specific member data

```
SUNDomEigEstimator_Ops ops
```

A virtual table of dominant eigenvalue estimator operations provided by a specific implementation

```
SUNContext sunctx
```

The SUNDIALS simulation context

The virtual table structure is defined as

```
typedef struct SUNDomEigEstimator_Ops_ *SUNDomEigEstimator_Ops
```

```
struct SUNDomEigEstimator_Ops_
```

The structure defining *SUNDomEigEstimator* operations.

```
SUNErrCode (*setatimes)(SUNDomEigEstimator, void*, SUNATimesFn)
```

The function implementing *SUNDomEigEstimator\_SetATimes()*

```
SUNErrCode (*setmaxiters)(SUNDomEigEstimator, int)
```

The function implementing *SUNDomEigEstimator\_SetMaxIters()*

```
SUNErrCode (*setnumpreprocessiters)(SUNDomEigEstimator, int)
```

The function implementing *SUNDomEigEstimator\_SetNumPreprocessIters()*

```
SUNErrCode (*setreltol)(SUNDomEigEstimator, sunrealtype)
```

The function implementing *SUNDomEigEstimator\_SetRelTol()*

*SUNErrCode* (\***setinitialguess**)(*SUNDomEigEstimator*, *N\_Vector*)

The function implementing *SUNDomEigEstimator\_SetInitialGuess()*

*SUNErrCode* (\***initialize**)(*SUNDomEigEstimator*)

The function implementing *SUNDomEigEstimator\_Initialize()*

*SUNErrCode* (\***estimate**)(*SUNDomEigEstimator*, *sunrealtype\**, *sunrealtype\**)

The function implementing *SUNDomEigEstimator\_Estimate()*

*sunrealtype* (\***getres**)(*SUNDomEigEstimator*)

The function implementing *SUNDomEigEstimator\_GetRes()*

int (\***getnumiters**)(*SUNDomEigEstimator*)

The function implementing *SUNDomEigEstimator\_GetNumIters()*

long int (\***getnumatimescalls**)(*SUNDomEigEstimator*)

The function implementing *SUNDomEigEstimator\_GetNumATimesCalls()*

*SUNErrCode* (\***write**)(*SUNDomEigEstimator*, FILE\*)

The function implementing *SUNDomEigEstimator\_Write()*

*SUNErrCode* (\***destroy**)(*SUNDomEigEstimator\**)

The function implementing *SUNDomEigEstimator\_Destroy()*

The generic *SUNDomEigEst* class defines and implements the dominant eigenvalue estimator operations defined in §12.2.1 – §12.2.3. These routines are in fact only wrappers to the dominant eigenvalue estimator operations defined by a particular *SUNDomEigEst* implementation, which are accessed through the *ops* field of the *SUNDomEigEstimator* structure. To illustrate this point we show below the implementation of a typical dominant eigenvalue estimator operation from the *SUNDomEigEstimator* base class, namely *SUNDomEigEstimator\_Initialize()*, that initializes a *SUNDomEigEstimator* object for use after it has been created and configured, and returns a flag denoting a successful or failed operation:

```
SUNErrCode SUNDomEigEstimator_Initialize(SUNDomEigEstimator DEE)
{
    return (DEE->ops->initialize(DEE));
}
```

Additionally, a *SUNDomEigEstimator* implementation *may* do the following:

- Define and implement additional user-callable “set” routines acting on the *SUNDomEigEstimator*, e.g., for setting various configuration options to tune the dominant eigenvalue estimator for a particular problem.
- Provide additional user-callable “get” routines acting on the *SUNDomEigEstimator* object, e.g., for returning various estimator statistics.

## 12.3 SUNDIALS modules *SUNDomEigEstimator* interface

In Table 12.1, we list the *SUNDomEigEst* module functions used within SUNDIALS packages. We emphasize that the user does not need to know detailed usage of dominant eigenvalue estimator functions by a SUNDIALS package in order to use it. The information is presented as an implementation detail for the interested reader.

Table 12.1: List of `SUNDomEigEst` functions called by a SUNDIALS module dominant eigenvalue estimator interface. Functions marked with “X” are required; functions marked with “O” are only called if they are non-NULL and functions marked with “N/A” are not applicable in the `SUNDomEigEstimator` implementation that is being used.

Routine	Power Iteration	Arnoldi Iteration
<code>SUNDomEigEstimator_SetATimes()</code>	X	X
<code>SUNDomEigEstimator_SetMaxIters()</code> <sup>1</sup>	O	N/A
<code>SUNDomEigEstimator_SetNumPreprocessIters()</code>	O	O
<code>SUNDomEigEstimator_SetRelTol()</code> <sup>1</sup>	O	N/A
<code>SUNDomEigEstimator_SetInitialGuess()</code>	O	O
<code>SUNDomEigEstimator_Initialize()</code>	X	X
<code>SUNDomEigEstimator_Estimate()</code>	X	X
<code>SUNDomEigEstimator_GetRes()</code> <sup>2</sup>	O	O
<code>SUNDomEigEstimator_GetNumIters()</code>	O	O
<code>SUNDomEigEstimator_GetNumATimesCalls()</code>	O	O
<code>SUNDomEigEstimator_Write()</code>	O	O
<code>SUNDomEigEstimator_Destroy()</code> <sup>3</sup>		

Notes:

1. `SUNDomEigEstimator_SetMaxIters()` and `SUNDomEigEstimator_SetRelTol()` might or might not be required depending on `SUNDomEigEstimator` implementation that is being used. These operations should be left as NULL if it is not applicable for an estimator.
2. Although `SUNDomEigEstimator_GetRes()` is optional, if it is not implemented by the `SUNDomEigEstimator` then the interface will consider all estimates a being *exact*.
3. Although the interface does not call `SUNDomEigEstimator_Destroy()` directly, this routine should be available for users to call when cleaning up from a simulation.

## 12.4 The `SUNDomEigEstimator_Power` Module

Added in version 7.5.0.

The `SUNDomEigEstimator_Power` implementation of the `SUNDomEigEstimator` class performs the Power Iteration (PI) method [123]; this is an iterative dominant eigenvalue estimator that is designed to be compatible with any `N_Vector` implementation that supports a minimal subset of operations (`N_VClone()`, `N_VDotProd()`, `N_VScale()`, and `N_VDestroy()`).

Power iteration is useful for large, sparse matrices whose dominant eigenvalue is real-valued and has algebraic multiplicity one. The algorithm starts with a non-zero vector  $\mathbf{v}_0$ . It then iteratively updates this via

$$\mathbf{v}_{k+1} = \frac{A\mathbf{v}_k}{\|A\mathbf{v}_k\|},$$

where  $\|\cdot\|$  denotes the Euclidean norm. Over successive iterations,  $\mathbf{v}_k$  converges to the eigenvector corresponding to the dominant eigenvalue of  $A$ . At each step, the corresponding eigenvalue can be approximated using the Rayleigh quotient

$$\lambda_k = \frac{\mathbf{v}_k^T A \mathbf{v}_k}{\|\mathbf{v}_k\|^2}.$$

The iteration continues until the two successive eigenvalue approximations are relatively close enough to one another. That is, for some *relative tolerance*.

Power iteration works for the matrices that have a **real** dominant eigenvalue. If it is strictly greater than all others (in magnitude), convergence is guaranteed. The speed of convergence depends on the ratios of the magnitude of the first two dominant eigenvalues.

The matrix  $A$  is not required explicitly; only a routine that provides the matrix-vector product  $Av$  is required. Also, PI requires a fixed amount of memory regardless of the number of iterations.

### 12.4.1 SUNDomEigEstimator\_Power Usage

To use `SUNDomEigEstimator_Arnoldi` include the header file `sundomeigest/sundomeigest_power.h`, and link to the library `libsundials_sundomeigestpower`.

The module `SUNDomEigEstimator_Power` provides the following user-callable routines:

*SUNDomEigEstimator* **SUNDomEigEstimator\_Power**(*N\_Vector* q, long int max\_iters, *sunrealtype* rel\_tol, *SUNContext* sunctx)

This constructor function creates and allocates memory for the Power iteration implementation of a *SUNDomEigEstimator*.

Consistency checks are performed to ensure the input vector is non-zero and supplies the necessary operations.

#### Parameters

- **q** – the initial guess for the dominant eigenvector; this should not be a non-dominant eigenvector of the Jacobian.
- **max\_iters** – maximum number of iterations (default 100). Supplying a value  $\leq 0$  will result in using the default value. Although this default number is not high for large matrices, it is reasonable since (1) most solvers do not need too tight tolerances and consider a safety factor, and (2) an early (less costly) termination will be a good indicator whether the power iteration is compatible.
- **rel\_tol** – relative tolerance for convergence checks (default 0.005). A value  $\leq 0$  will result in the default value. The default has been found to small enough for many internal applications.
- **sunctx** – the *SUNContext* object.

#### Returns

If successful, a *SUNDomEigEstimator* otherwise NULL.

#### Note

When used in a time-dependent context, the initial guess supplied to the constructor, q, is used only for the first *SUNDomEigEstimator\_Estimate()* call and is overwritten with the result of the next to last Power iteration from the most recent *SUNDomEigEstimator\_Estimate()* call. This new value is used as the initial guess for subsequent estimates.

The initial guess can be reset with *SUNDomEigEstimator\_SetInitialGuess()*.

### 12.4.2 SUNDomEigEstimator\_Power Description

The SUNDomEigEstimator\_Power module defines the *content* field of a SUNDomEigEstimator to be the following structure:

```
struct SUNDomEigEstimatorContent_Power_ {
    SUNATimesFn ATimes;
    void* ATdata;
    N_Vector* V;
    N_Vector q;
    int num_warmups;
    long int max_iters;
    long int num_iters;
    long int num_ATimes;
    sunrealtype rel_tol;
    sunrealtype res;
};
```

These entries of the *content* field contain the following information:

- *ATimes* - function pointer to perform the product  $Av$ ,
- *ATData* - pointer to structure for *ATimes*,
- *V*, *q* - *N\_Vector* used for workspace by the PI algorithm.
- *num\_warmups* - number of preprocessing iterations (default is 100),
- *max\_iters* - maximum number of iterations (default is 100),
- *num\_iters* - number of iterations (preprocessing and estimation) in the last *SUNDomEigEstimator\_Estimate()* call,
- *num\_ATimes* - number of calls to the *ATimes* function,
- *rel\_tol* - relative tolerance for the convergence criteria (default is 0.005),
- *res* - the residual from the last *SUNDomEigEstimator\_Estimate()* call.

This estimator is constructed to perform the following operations:

- During construction all *N\_Vector* estimator data is allocated, with vectors cloned from a template *N\_Vector* that is input, and default generic estimator parameters are set.
- User-facing “set” routines may be called to modify default estimator parameters.
- SUNDIALS packages will call *SUNDomEigEstimator\_SetATimes()* to supply the *ATimes* function pointer and the related data *ATData*.
- In *SUNDomEigEstimator\_Initialize()*, the estimator parameters are checked for validity and the initial eigenvector is normalized.
- In *SUNDomEigEstimator\_Estimate()*, the initial nonzero vector  $q_0$  is preprocessed with some fixed number of Power iterations,

$$q_1 = \frac{Aq_0}{\|Aq_0\|} \quad \cdots \quad q_k = \frac{Aq_{k-1}}{\|Aq_{k-1}\|},$$

(see *LSRKStepSetNumDomEigEstInitPreprocessIters()* and *LSRKStepSetNumDomEigEstPreprocessIters()* for setting the number of preprocessing iterations) before computing the estimate.

The SUNDomEigEstimator\_Power module defines implementations of all dominant eigenvalue estimator operations listed in §12.2:

- `SUNDomEigEstimator_SetATimes_Power`
- `SUNDomEigEstimator_SetMaxIters_Power`
- `SUNDomEigEstimator_SetNumPreprocessIters_Power`
- `SUNDomEigEstimator_SetRelTol_Power`
- `SUNDomEigEstimator_Initialize_Power`
- `SUNDomEigEstimator_Estimate_Power`
- `SUNDomEigEstimator_SetInitialGuess_Power`
- `SUNDomEigEstimator_GetRes_Power`
- `SUNDomEigEstimator_GetNumIters_Power`
- `SUNDomEigEstimator_GetNumATimesCalls_Power`
- `SUNDomEigEstimator_Write_Power`
- `SUNDomEigEstimator_Destroy_Power`

## 12.5 The `SUNDomEigEstimator_Arnoldi` Module

Added in version 7.5.0.

The `SUNDomEigEstimator_Arnoldi` implementation of the `SUNDomEigEstimator` class performs the Arnoldi Iteration method [12]; this is an iterative dominant eigenvalue estimator that is designed to be compatible with any `N_Vector` implementation that supports a minimal subset of operations (`N_VClone()`, `N_VDotProd()`, `N_VScale()`, and `N_VDestroy()`).

Arnoldi iteration is particularly effective for large, sparse matrices where only the dominant eigenvalue is needed. It constructs an orthonormal basis of the Krylov subspace

$$\mathcal{K}_m(A, \mathbf{v}) = \text{span}\{\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \dots, A^{m-1}\mathbf{v}\}$$

using the Gram-Schmidt process. The matrix  $A$  is projected onto this subspace to form a small upper Hessenberg matrix  $H_m$ . The eigenvalues of  $H_m$  approximate some of the eigenvalues of  $A$ ; the dominant eigenvalue of  $A$  is well-approximated by the dominant eigenvalue of  $H_m$ .

Arnoldi iteration works for matrices with both real and complex eigenvalues. It supports estimations with a user-specified fixed Krylov subspace dimension (at least 3). While the choice of dimension results in a prefixed amount of memory, it strictly determines the quality of the estimate. To improve the estimation accuracy, we have found that preprocessing with a number of Power iterations is particularly useful. This operation is free from any additional memory requirement and is further explained below.

The matrix  $A$  is not required explicitly; only a routine that provides an approximation of the matrix-vector product,  $A\mathbf{v}$ , is required.

### 12.5.1 `SUNDomEigEstimator_Arnoldi` Usage

To use `SUNDomEigEstimator_Arnoldi` include the header file `sundomeigest/sundomeigest_arnoldi.h`, and link to the library `libsundials_sundomeigestarnoldi`.

The module `SUNDomEigEstimator_Arnoldi` provides the following user-callable routines:

*SUNDomEigEstimator* **SUNDomEigEstimator\_Arnoldi**(*N\_Vector* q, int kry\_dim, *SUNContext* sunctx);

This constructor function creates and allocates memory for the Arnoldi iteration implementation of a *SUNDomEigEstimator*.

Consistency checks are performed to ensure the input vector is non-zero and supplies the necessary operations.

#### Parameters

- **q** – the initial guess for the dominant eigenvector; this should not be a non-dominant eigenvector of the Jacobian.
- **kry\_dim** – the dimension of the Krylov subspace (default 3). A value  $\leq 2$  will result in using default value. This default is chosen to minimize the memory footprint.
- **sunctx** – the *SUNContext* object.

#### Returns

If successful, a *SUNDomEigEstimator* otherwise NULL.

#### Note

When used in a time-dependent context, the initial guess supplied to the constructor, q, is used only in the first *SUNDomEigEstimator\_Estimate()* call and is overwritten with the result of the most recent preprocessing iterations (see *SUNDomEigEstimator\_SetNumPreprocessIters()*). As an initial guess too close to the dominant eigenvector may cause a breakdown in the Gram–Schmidt process within the Arnoldi iteration, users should account for this when setting the number of initial and subsequent preprocessing iterations (e.g., with LSRKStep see *LSRKStepSetNumDomEigEstInitPreprocessIters()* and *LSRKStepSetNumDomEigEstPreprocessIters()*).

The initial guess can be reset with *SUNDomEigEstimator\_SetInitialGuess()*.

## 12.5.2 SUNDomEigEstimator\_Arnoldi Description

The *SUNDomEigEstimator\_Arnoldi* module defines the *content* field of a *SUNDomEigEstimator* to be the following structure:

```
struct SUNDomEigEstimatorContent_Arnoldi_ {
    SUNATimesFn ATimes;
    void* ATdata;
    N_Vector* V;
    N_Vector q;
    int kry_dim;
    int num_warmups;
    long int num_iters;
    long int num_ATimes;
    sunrealtype* LAPACK_A;
    sunrealtype* LAPACK_wr;
    sunrealtype* LAPACK_wi;
    sunrealtype* LAPACK_work;
    sunindextype LAPACK_lwork;
    sunrealtype** LAPACK_arr;
    sunrealtype** Hes;
};
```

These entries of the *content* field contain the following information:

- `ATimes` - function pointer to perform the product  $Av$ ,
- `ATData` - pointer to structure for `ATimes`,
- `V`, `q` - vectors used for workspace by the Arnoldi algorithm.
- `kry_dim` - dimension of Krylov subspaces (default is 3),
- `num_warmups` - number of preprocessing iterations (default is 100),
- `num_iters` - number of iterations (preprocessing and estimation) in the last `SUNDomEigEstimator_Estimate()` call,
- `num_ATimes` - number of calls to the `ATimes` function,
- `LAPACK_A`, `LAPACK_wr`, `LAPACK_wi`, `LAPACK_work` - `sunrealtype` used for workspace by LAPACK,
- `LAPACK_lwork` - the size of the `LAPACK_work` requested by LAPACK,
- `LAPACK_arr` - storage for the estimated dominant eigenvalues,
- `Hes` - Hessenberg matrix,

This estimator is constructed to perform the following operations:

- During construction all `N_Vector` estimator data is allocated, with vectors cloned from a template `N_Vector` that is input, and default generic estimator parameters are set.
- User-facing “set” routines may be called to modify default estimator parameters.
- SUNDIALS packages will call `SUNDomEigEstimator_SetATimes()` to supply the `ATimes` function pointer and the related data `ATData`.
- In `SUNDomEigEstimator_Initialize()`, the estimator parameters are checked for validity and the remaining Arnoldi estimator memory such as LAPACK workspace is allocated.
- In `SUNDomEigEstimator_Estimate()`, the initial nonzero vector  $q_0$  is preprocessed with some fixed number of Power iterations,

$$q_1 = \frac{Aq_0}{\|Aq_0\|} \quad \cdots \quad q_k = \frac{Aq_{k-1}}{\|Aq_{k-1}\|},$$

(see `LSRKStepSetNumDomEigEstInitPreprocessIters()` and `LSRKStepSetNumDomEigEstPreprocessIters()` for setting the number of preprocessing iterations). Then, the Arnoldi iteration is performed to compute the estimate.

The `SUNDomEigEstimator_Arnoldi` module defines implementations of all dominant eigenvalue estimator operations listed in §12.2:

- `SUNDomEigEstimator_SetATimes_Arnoldi`
- `SUNDomEigEstimator_SetNumPreprocessIters_Arnoldi`
- `SUNDomEigEstimator_Initialize_Arnoldi`
- `SUNDomEigEstimator_Estimate_Arnoldi`
- `SUNDomEigEstimator_GetNumIters_Arnoldi`
- `SUNDomEigEstimator_GetNumATimesCalls_Arnoldi`
- `SUNDomEigEstimator_Write_Arnoldi`
- `SUNDomEigEstimator_Destroy_Arnoldi`



## Chapter 13

# Time Step Adaptivity Controllers

The SUNDIALS library comes packaged with a variety of *SUNAdaptController* implementations, designed to support various forms of error-based time step adaptivity within SUNDIALS time integrators. To support applications that may want to adjust the controller parameters or provide their own implementations, SUNDIALS defines the *SUNAdaptController* base class, along with a variety of default implementations.

### 13.1 The SUNAdaptController API

Added in version 6.7.0.

Changed in version 7.2.0: Added support multirate time step adaptivity controllers

The *SUNAdaptController* base class provides a common API for accuracy-based adaptivity controllers to be used by SUNDIALS integrators. These controllers estimate step sizes (among other things) such that the next step solution satisfies a desired temporal accuracy, while striving to maximize computational efficiency. We note that in the descriptions below, we frequently use *dsm* to represent temporal error. This is **not** the raw temporal error estimate; instead, it is a norm of the temporal error estimate after scaling by the user-supplied accuracy tolerances (see (2.24)),

$$dsm = \left( \frac{1}{N} \sum_{i=1}^N \left( \frac{\text{error}_i}{\text{rtol} \cdot |y_{n-1,i}| + \text{atol}_i} \right)^2 \right)^{1/2}.$$

Thus *dsm* values below one represent errors estimated to be more accurate than needed, whereas errors above one are considered to be larger than allowable.

The *SUNAdaptController* class is modeled after SUNDIALS' other object-oriented classes, in that this class contains a pointer to an implementation-specific *content*, an *ops* structure with generic controller operations, and a *SUNContext* object.

A *SUNAdaptController* is a pointer to the *\_generic\_SUNAdaptController* structure:

```
typedef struct _generic_SUNAdaptController *SUNAdaptController
```

```
struct _generic_SUNAdaptController
```

```
void *content
```

Pointer to the controller-specific member data

```
SUNAdaptController_Ops ops;
```

A virtual table of controller operations provided by a specific implementation

***SUNContext* sunctx**

The SUNDIALS simulation context

The virtual table structure is defined as

```
typedef struct _generic_SUNAdaptController_Ops *SUNAdaptController_Ops
```

```
struct _generic_SUNAdaptController_Ops
```

The structure defining *SUNAdaptController* operations.

*SUNAdaptController\_Type* (\***gettype**)(*SUNAdaptController* C)

The function implementing *SUNAdaptController\_GetType*()

*SUNErrorCode* (\***destroy**)(*SUNAdaptController* C)

The function implementing *SUNAdaptController\_Destroy*()

*SUNErrorCode* (\***estimatestep**)(*SUNAdaptController* C, *sunrealtype* h, int p, *sunrealtype* dsm, *sunrealtype* \*hnew)

The function implementing *SUNAdaptController\_EstimateStep*()

*SUNErrorCode* (\***estimatesteptol**)(*SUNAdaptController* C, *sunrealtype* H, *sunrealtype* tolfac, int P, *sunrealtype* DSM, *sunrealtype* dsm, *sunrealtype* \*Hnew, *sunrealtype* \*tolfacnew)

The function implementing *SUNAdaptController\_EstimateStepTol*()

Added in version 7.2.0.

*SUNErrorCode* (\***reset**)(*SUNAdaptController* C)

The function implementing *SUNAdaptController\_Reset*()

*SUNErrorCode* (\***setoptions**)(*SUNAdaptController* C, const char \*Cid, const char \*file\_name, int argc, char \*argv[])

The function implementing *SUNAdaptController\_SetOptions*()

Added in version 7.5.0.

*SUNErrorCode* (\***setdefaults**)(*SUNAdaptController* C)

The function implementing *SUNAdaptController\_SetDefaults*()

*SUNErrorCode* (\***write**)(*SUNAdaptController* C, FILE \*fptr)

The function implementing *SUNAdaptController\_Write*()

*SUNErrorCode* (\***seterrorbias**)(*SUNAdaptController* C, *sunrealtype* bias)

The function implementing *SUNAdaptController\_SetErrorBias*()

*SUNErrorCode* (\***updateh**)(*SUNAdaptController* C, *sunrealtype* h, *sunrealtype* dsm)

The function implementing *SUNAdaptController\_UpdateH*()

*SUNErrorCode* (\***updatemritol**)(*SUNAdaptController* C, *sunrealtype* H, *sunrealtype* tolfac, *sunrealtype* DSM, *sunrealtype* dsm)

The function implementing *SUNAdaptController\_UpdateMRIHTol*()

Added in version 7.2.0.

*SUNErrorCode* (\***space**)(*SUNAdaptController* C, long int \*lenrw, long int \*leniw)

The function implementing *SUNAdaptController\_Space*()

### 13.1.1 SUNAdaptController Types

The time integrators in SUNDIALS adapt a variety of parameters to achieve accurate and efficient computations. To this end, each SUNAdaptController implementation should note its type, so that integrators will understand the types of adaptivity that the controller is designed to perform. These are encoded in the following set of SUNAdaptController types:

enum **SUNAdaptController\_Type**

The enumerated type *SUNAdaptController\_Type* defines the enumeration constants for SUNDIALS error controller types

enumerator **SUN\_ADAPTCONTROLLER\_NONE**

Empty object that performs no control.

enumerator **SUN\_ADAPTCONTROLLER\_H**

Controls a single-rate step size.

enumerator **SUN\_ADAPTCONTROLLER\_MRI\_H\_TOL**

Controls both a slow time step and a tolerance factor to apply on the next-faster time scale within a multirate simulation that has an arbitrary number of time scales.

Added in version 7.2.0.

### 13.1.2 SUNAdaptController Operations

The base SUNAdaptController class defines and implements all SUNAdaptController functions. Most of these routines are merely wrappers for the operations defined by a particular SUNAdaptController implementation, which are accessed through the *ops* field of the SUNAdaptController structure. The base SUNAdaptController class provides the constructor

*SUNAdaptController* **SUNAdaptController\_NewEmpty**(*SUNContext* sunctx)

This function allocates a new generic SUNAdaptController object and initializes its content pointer and the function pointers in the operations structure to NULL.

#### Parameters

- **sunctx** – the *SUNContext* object (see §4.2)

#### Returns

If successful, a generic *SUNAdaptController* object. If unsuccessful, a NULL pointer will be returned.

Each of the following methods are *optional* for any specific SUNAdaptController implementation, however some may be required based on the implementation's *SUNAdaptController\_Type* (see Section §13.1.1). We note these requirements below. Additionally, we note the behavior of the base SUNAdaptController methods when they perform an action other than only a successful return.

void **SUNAdaptController\_DestroyEmpty**(*SUNAdaptController* C)

This routine frees the generic SUNAdaptController object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will free it as well.

#### Parameters

- **C** – the *SUNAdaptController* object.

#### Returns

*SUNErrCode* indicating success or failure.

*SUNAdaptController\_Type* **SUNAdaptController\_GetType**(*SUNAdaptController* C)

Returns the type identifier for the controller *C*. Returned values are given in Section §13.1.1

**Parameters**

- *C* – the *SUNAdaptController* object.

**Returns**

*SUNAdaptController\_Type* type identifier.

*SUNErrCode* **SUNAdaptController\_Destroy**(*SUNAdaptController* C)

Deallocates the controller *C*. If this method is not provided by the implementation, the base class method will free both the *content* and *ops* objects – this should be sufficient unless a controller implementation performs dynamic memory allocation of its own (note that the SUNDIALS-provided *SUNAdaptController* implementations do not need to supply this routine).

**Parameters**

- *C* – the *SUNAdaptController* object.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNAdaptController\_EstimateStep**(*SUNAdaptController* C, *sunrealtype* h, int p, *sunrealtype* dsm, *sunrealtype* \*hnew)

Estimates a single-rate step size. This routine is required for controllers of type `SUN_ADAPTCONTROLLER_H`. If this is not provided by the implementation, the base class method will set *\*hnew* = *h* and return.

**Parameters**

- *C* – the *SUNAdaptController* object.
- *h* – the step size from the previous step attempt.
- *p* – the current order of accuracy for the time integration method.
- *dsm* – the local temporal estimate from the previous step attempt.
- *hnew* – (output) the estimated step size.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNAdaptController\_EstimateStepTol**(*SUNAdaptController* C, *sunrealtype* H, *sunrealtype* tolfac, int P, *sunrealtype* DSM, *sunrealtype* dsm, *sunrealtype* \*Hnew, *sunrealtype* \*tolfacnew)

Estimates a slow step size and a fast tolerance multiplication factor for two adjacent time scales within a multirate application.

This routine is required for controllers of type `:c:enumerator`SUN_ADAPTCONTROLLER_MRI_H_TOL``. If the current time scale has relative tolerance *rtol*, then the next-faster time scale will be called with relative tolerance *tolfac* \* *rtol*. If this is not provided by the implementation, the base class method will set *\*Hnew* = *H* and *\*tolfacnew* = *tolfac* and return.

**Parameters**

- *C* – the *SUNAdaptController* object.
- *H* – the slow step size from the previous step attempt.
- *tolfac* – the current relative tolerance factor for the next-faster time scale.
- *P* – the current order of accuracy for the slow time scale integration method.

- **DSM** – the slow time scale local temporal estimate from the previous step attempt.
- **dsm** – the fast time scale local temporal estimate from the previous step attempt.
- **Hnew** – (output) the estimated slow step size.
- **tolfacnew** – (output) the estimated relative tolerance factor.

**Returns**

*SUNErrCode* indicating success or failure.

Added in version 7.2.0.

*SUNErrCode* **SUNAdaptController\_Reset**(*SUNAdaptController* C)

Resets the controller to its initial state, e.g., if it stores a small number of previous dsm or h values.

**Parameters**

- **C** – the *SUNAdaptController* object.

**Returns**

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNAdaptController\_SetOptions**(*SUNAdaptController* C, const char \*Cid, const char \*file\_name, int argc, char \*argv[])

Sets SUNAdaptController options from an array of strings or a file.

**Parameters**

- **C** – the *SUNAdaptController* object.
- **Cid** – the prefix for options to read. The default is “sunadaptcontroller”.
- **file\_name** – the name of a file containing options to read. If this is NULL or an empty string, “”, then no file is read.
- **argc** – number of command-line arguments passed to executable.
- **argv** – an array of strings containing the options to set and their values.

**Returns**

*SUNErrCode* indicating success or failure.

**Note**

The argc and argv arguments are typically those supplied to the user’s main routine however, this is not required. The inputs are left unchanged by *SUNAdaptController\_SetOptions()*.

If the Cid argument is NULL then the default prefix, sunadaptcontroller, must be used for all SUNAdaptController options. Whether Cid is supplied or not, a “.” must be used to separate an option key from the prefix. For example, when using the default Cid, the option sunadaptcontroller.error\_bias followed by the value can be used to set the error bias factor. When using a combination of SUNAdaptController objects (e.g., within MRISStep, SplittingStep or ForcingStep), it is recommended that users call *SUNAdaptController\_SetOptions()* for each controller using distinct Cid inputs, so that each controller can be configured separately.

SUNAdaptController options set via *SUNAdaptController\_SetOptions()* will overwrite any previously-set values. Options are set in the order they are given in argv and, if an option with the same prefix appears multiple times in argv, the value of the last occurrence will be used.

The supported option names are noted within the documentation for the corresponding SUNAdaptController functions. For options that take a *sunbooleantype* as input, use 1 to indicate true and 0 for false.

**Warning**

This function is not available in the Fortran interface.

File-based options are not yet supported, so the `file_name` argument should be set to either `NULL` or the empty string `""`.

Added in version 7.5.0.

*SUNErrCode* **SUNAdaptController\_SetDefaults**(*SUNAdaptController* C)

Sets the controller parameters to their default values.

**Parameters**

- **C** – the *SUNAdaptController* object.

**Returns**

*SUNErrCode* indicating success or failure.

**Note**

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.defaults”.

*SUNErrCode* **SUNAdaptController\_Write**(*SUNAdaptController* C, FILE \*fptr)

Writes all controller parameters to the indicated file pointer.

**Parameters**

- **C** – the *SUNAdaptController* object.
- **fptr** – the output stream to write the parameters to.

**Returns**

*SUNErrCode* indicating success or failure.

**Note**

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.write\_parameters”.

*SUNErrCode* **SUNAdaptController\_SetErrorBias**(*SUNAdaptController* C, *sunrealtype* bias)

Sets an error bias factor for scaling the local error factors. This is typically used to slightly exaggerate the temporal error during the estimation process, leading to a more conservative estimated step size.

**Parameters**

- **C** – the *SUNAdaptController* object.
- **bias** – the error bias factor – an input  $\leq 0$  indicates to use the default value for the controller.

**Returns**

*SUNErrCode* indicating success or failure.

**Note**

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.error\_bias”.

*SUNErrCode* **SUNAdaptController\_UpdateH**(*SUNAdaptController* C, *sunrealtype* h, *sunrealtype* dsm)

Notifies a controller of type `SUN_ADAPTCONTROLLER_H` that a successful time step was taken with stepsize  $h$  and local error factor  $dsm$ , indicating that these can be saved for subsequent controller functions. This is typically relevant for controllers that store a history of either step sizes or error estimates for performing the estimation process.

#### Parameters

- **C** – the *SUNAdaptController* object.
- **h** – the successful step size.
- **dsm** – the successful temporal error estimate.

#### Returns

*SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNAdaptController\_UpdateMRIHTol**(*SUNAdaptController* C, *sunrealtype* H, *sunrealtype* tolfac, *sunrealtype* DSM, *sunrealtype* dsm)

Notifies a controller of type `SUN_ADAPTCONTROLLER_MRI_H_TOL` that a successful time step was taken with slow stepsize  $H$  and fast relative tolerance factor  $\text{tolfac}$ , and that the step had slow and fast local error factors  $DSM$  and  $dsm$ , indicating that these can be saved for subsequent controller functions. This is typically relevant for controllers that store a history of either step sizes or error estimates for performing the estimation process.

#### Parameters

- **C** – the *SUNAdaptController* object.
- **H** – the successful slow step size.
- **tolfac** – the successful fast time scale relative tolerance factor.
- **DSM** – the successful slow temporal error estimate.
- **dsm** – the successful fast temporal error estimate.

#### Returns

*SUNErrCode* indicating success or failure.

Added in version 7.2.0.

*SUNErrCode* **SUNAdaptController\_Space**(*SUNAdaptController* C, long int \*lenrw, long int \*leniw)

Informative routine that returns the memory requirements of the *SUNAdaptController* object.

#### Parameters

- **C** – the *SUNAdaptController* object..
- **lenrw** – (output) number of *sunrealtype* words stored in the controller.
- **leniw** – (output) number of *sunindextype* words stored in the controller. This may also include pointers, *int* and *long int* words.

#### Returns

*SUNErrCode* indicating success or failure.

Deprecated since version 7.3.0: Work space functions will be removed in version 8.0.0.

### 13.1.3 C/C++ API Usage

Specific SUNDIALS adaptivity controller modules can be used in C and C++ programs by including the corresponding header file for that module, e.g. `sunadaptcontroller/sunadaptcontroller_XYZ.h`.

Example usage (here SUNAdaptController\_XYZ is a placeholder for an actual SUNAdaptController constructor):

```
#include <stdio.h>
#include <stdlib.h>
#include <sundials/sundials_context.h>
#include <sundials/sundials_types.h>
#include <sunadaptcontroller/sunadaptcontroller_XYZ.h>

int main()
{
    /* Create a SUNContext object */
    SUNContext sunctx = ...;

    /* Create a SUNAdaptController object */
    SUNAdaptController C = SUNAdaptController_XYZ(sunctx);

    /* Use the control object */

    /* Destroy the control object */
    retval = SUNAdaptController_Destroy(C);

    return 0;
}
```

## 13.2 The SUNAdaptController\_Soderlind Module

The Soderlind implementation of the SUNAdaptController class, SUNAdaptController\_Soderlind, implements a general structure for temporal control proposed by G. Soderlind in [103], [104], and [105]. This controller has the form

$$h' = h_n \varepsilon_n^{-k_1/(p+1)} \varepsilon_{n-1}^{-k_2/(p+1)} \varepsilon_{n-2}^{-k_3/(p+1)} \left( \frac{h_n}{h_{n-1}} \right)^{k_4} \left( \frac{h_{n-1}}{h_{n-2}} \right)^{k_5}$$

with default parameter values  $k_1 = 1.25$ ,  $k_2 = 0.5$ ,  $k_3 = -0.75$ ,  $k_4 = 0.25$ , and  $k_5 = 0.75$ , where  $p$  is the global order of the time integration method. If there is insufficient history of past time steps and errors, i.e., on the first or second time step, an I controller is used.

The SUNAdaptController\_Soderlind controller is implemented as a derived SUNAdaptController class, and defines its *content* field as:

```
struct _SUNAdaptControllerContent_Soderlind {
    sunrealtype k1;
    sunrealtype k2;
    sunrealtype k3;
    sunrealtype k4;
    sunrealtype k5;
    sunrealtype bias;
    sunrealtype ep;
    sunrealtype epp;
    sunrealtype hp;
    sunrealtype hpp;
    int firststeps;
    int historysize;
};
```



These entries of the *content* field contain the following information:

- **k1, k2, k3, k4, k5** - controller parameters above.
- **bias** - error bias factor, that converts from an input temporal error estimate via  $\varepsilon = \text{bias} * \text{dsm}$ .
- **ep, epp** - storage for the two previous error estimates,  $\varepsilon_{n-1}$  and  $\varepsilon_{n-2}$ .
- **hp, hpp** - storage for the previous two step sizes,  $h_{n-1}$  and  $h_{n-2}$ .
- **firststeps** - counter to handle first two steps (where previous step sizes and errors are unavailable).
- **historysize** - number of past step sizes or errors needed.

The header file to be included when using this module is `sunadaptcontroller/sunadaptcontroller_soderlind.h`.

We note that through appropriate selection of the parameters  $k_*$ , this controller may create a wide range of proposed temporal adaptivity controllers, including the  $H_0321$ , I, PI, PID, as well as Gustafsson's explicit and implicit controllers, [52] and [53], among others. As a convenience, utility routines to create a variety of these controllers and set their parameters (as special cases of `SUNAdaptController_Soderlind`) are provided.

The `SUNAdaptController_Soderlind` class provides implementations of all operations relevant to a `SUN_ADAPTCONTROLLER_H` controller listed in §13.1.2. This class also provides the following additional user-callable routines:

*SUNAdaptController* **SUNAdaptController\_Soderlind**(*SUNContext* sunctx)

This constructor creates and allocates memory for a `SUNAdaptController_Soderlind` object, and inserts its default parameters.

#### Parameters

- **sunctx** – the current *SUNContext* object.

#### Returns

if successful, a usable *SUNAdaptController* object; otherwise it will return `NULL`.

Usage:

```
SUNAdaptController C = SUNAdaptController_Soderlind(sunctx);
```

*SUNErrCode* **SUNAdaptController\_SetParams\_Soderlind**(*SUNAdaptController* C, *sunrealtype* k1, *sunrealtype* k2, *sunrealtype* k3, *sunrealtype* k4, *sunrealtype* k5)

This user-callable function provides control over the relevant parameters above. This should be called *before* the time integrator is called to evolve the problem.

#### Parameters

- **C** – the `SUNAdaptController_Soderlind` object.
- **k1** – parameter used within the controller time step estimate.
- **k2** – parameter used within the controller time step estimate.
- **k3** – parameter used within the controller time step estimate.
- **k4** – parameter used within the controller time step estimate.
- **k5** – parameter used within the controller time step estimate.

#### Returns

*SUNErrCode* indicating success or failure.

Usage:

```
/* Specify parameters for Soderlind's H_{0}312 controller */
retval = SUNAdaptController_SetParams_Soderlind(C, 0.25, 0.5, 0.25, -0.75, -0.25);
```

**Note**

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.params\_soderlind”.

*SUNAdaptController* **SUNAdaptController\_PID**(*SUNContext* sunctx)

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate a PID controller, and inserts its default parameters  $k_1 = 0.58$ ,  $k_2 = -0.21$ ,  $k_3 = 0.1$ , and  $k_4 = k_5 = 0$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Usage:

```
SUNAdaptController C = SUNAdaptController_PID(sunctx);
```

*SUNErrCode* **SUNAdaptController\_SetParams\_PID**(*SUNAdaptController* C, *sunrealtype* k1, *sunrealtype* k2, *sunrealtype* k3)

This user-callable function provides control over the relevant parameters above for a PID controller, setting  $k_4 = k_5 = 0$ . This should be called *before* the time integrator is called to evolve the problem.

**Parameters**

- **C** – the *SUNAdaptController\_Soderlind* object.
- **k1** – parameter used within the controller time step estimate.
- **k2** – parameter used within the controller time step estimate.
- **k3** – parameter used within the controller time step estimate.

**Returns**

*SUNErrCode* indicating success or failure.

Usage:

```
retval = SUNAdaptController_SetParams_PID(C, 0.58, -0.21, 0.1);
```

**Note**

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.params\_pid”.

*SUNAdaptController* **SUNAdaptController\_PI**(*SUNContext* sunctx)

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate a PI controller, and inserts its default parameters  $k_1 = 0.8$ ,  $k_2 = -0.31$ , and  $k_3 = k_4 = k_5 = 0$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Usage:

```
SUNAdaptController C = SUNAdaptController_PI(sunctx);
```

*SUNErrCode* **SUNAdaptController\_SetParams\_PI**(*SUNAdaptController* C, *sunrealtype* k1, *sunrealtype* k2)

This user-callable function provides control over the relevant parameters above for a PI controller, setting  $k_3 = k_4 = k_5 = 0$ . This should be called *before* the time integrator is called to evolve the problem.

**Parameters**

- **C** – the *SUNAdaptController\_Soderlind* object.
- **k1** – parameter used within the controller time step estimate.
- **k2** – parameter used within the controller time step estimate.

**Returns**

*SUNErrCode* indicating success or failure.

Usage:

```
retval = SUNAdaptController_SetParams_PI(C, 0.8, -0.31);
```

**Note**

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.params\_pi”.

*SUNAdaptController* **SUNAdaptController\_I**(*SUNContext* sunctx)

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate an I controller, and inserts its default parameters  $k_1 = 1.0$  and  $k_2 = k_3 = k_4 = k_5 = 0$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Usage:

```
SUNAdaptController C = SUNAdaptController_I(sunctx);
```

*SUNErrCode* **SUNAdaptController\_SetParams\_I**(*SUNAdaptController* C, *sunrealtype* k1)

This user-callable function provides control over the relevant parameters above for an I controller, setting  $k_2 = k_3 = k_4 = k_5 = 0$ . This should be called *before* the time integrator is called to evolve the problem.

**Parameters**

- **C** – the *SUNAdaptController\_Soderlind* object.
- **k1** – parameter used within the controller time step estimate.

**Returns**

*SUNErrCode* indicating success or failure.

Usage:

```
retval = SUNAdaptController_SetParams_I(C, 1.0);
```

**Note**

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.params\_i”.

*SUNAdaptController* **SUNAdaptController\_ExpGus**(*SUNContext* sunctx)

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate Gustafsson’s explicit controller [52], and inserts its default parameters  $k_1 = 0.635$ ,  $k_2 = -0.268$ , and  $k_3 = k_4 = k_5 = 0$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Usage:

```
SUNAdaptController C = SUNAdaptController_ExpGus(sunctx);
```

*SUNErrCode* **SUNAdaptController\_SetParams\_ExpGus**(*SUNAdaptController* C, *sunrealtype* k1\_hat, *sunrealtype* k2\_hat)

This user-callable function provides control over the relevant parameters above for the explicit Gustafsson controller, setting  $k_3 = k_4 = k_5 = 0$ . This should be called *before* the time integrator is called to evolve the problem.

**Note**

Gustafsson’s explicit controller has the form

$$h' = h_n \varepsilon_n^{-\hat{k}_1/(p+1)} \left( \frac{\varepsilon_n}{\varepsilon_{n-1}} \right)^{-\hat{k}_2/(p+1)}.$$

The inputs to this function correspond to the values of  $\hat{k}_1$  and  $\hat{k}_2$ , which are internally transformed into the Soderlind coefficients  $k_1 = \hat{k}_1 + \hat{k}_2$  and  $k_2 = -\hat{k}_2$ .

**Parameters**

- **C** – the *SUNAdaptController\_Soderlind* object.
- **k1\_hat** – parameter used within the explicit Gustafsson controller time step estimate.
- **k2\_hat** – parameter used within the explicit Gustafsson controller time step estimate.

**Returns**

*SUNErrCode* indicating success or failure.

Usage:

```
retval = SUNAdaptController_SetParams_ExpGus(C, 0.367, 0.268);
```

**Note**

This routine will be called by `SUNAdaptController_SetOptions()` when using the key “Cid.params\_-expgus”.

***SUNAdaptController* SUNAdaptController\_ImpGus(*SUNContext* sunctx)**

This constructor creates and allocates memory for a `SUNAdaptController_Soderlind` object, set up to replicate Gustafsson’s implicit controller [53], and inserts its default parameters  $k_1 = 1.93$ ,  $k_2 = -0.95$ ,  $k_4 = 1$ , and  $k_3 = k_5 = 0$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Usage:

```
SUNAdaptController C = SUNAdaptController_ImpGus(sunctx);
```

***SUNErrCode* SUNAdaptController\_SetParams\_ImpGus(*SUNAdaptController* C, *sunrealtype* k1\_hat, *sunrealtype* k2\_hat)**

This user-callable function provides control over the relevant parameters above for the implicit Gustafsson controller, setting  $k_4 = 1$  and  $k_3 = k_5 = 0$ . This should be called *before* the time integrator is called to evolve the problem.

**Note**

Gustafsson’s implicit controller has the form

$$h' = h_n \varepsilon_n^{-\hat{k}_1/(p+1)} \left( \frac{\varepsilon_n}{\varepsilon_{n-1}} \right)^{-\hat{k}_2/(p+1)} \left( \frac{h_n}{h_{n-1}} \right).$$

The inputs to this function correspond to the values of  $\hat{k}_1$  and  $\hat{k}_2$ , which are internally transformed into the Soderlind coefficients  $k_1 = \hat{k}_1 + \hat{k}_2$ ,  $k_2 = -\hat{k}_2$ , and  $k_4 = 1$ .

**Parameters**

- **C** – the `SUNAdaptController_Soderlind` object.
- **k1\_hat** – parameter used within the implicit Gustafsson controller time step estimate.
- **k2\_hat** – parameter used within the implicit Gustafsson controller time step estimate.

**Returns**

*SUNErrCode* indicating success or failure.

Usage:

```
retval = SUNAdaptController_SetParams_ImpGus(C, 0.98, 0.95);
```

**Note**

This routine will be called by `SUNAdaptController_SetOptions()` when using the key “Cid.params\_impqus”.

*SUNAdaptController* **SUNAdaptController\_H0211**(*SUNContext* sunctx)

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate the  $H_{0211}$  controller from [104], corresponding with the parameters  $k_1 = 0.5$ ,  $k_2 = 0.5$ ,  $k_4 = -0.5$ , and  $k_3 = k_5 = 0$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Added in version 7.3.0.

Usage:

```
SUNAdaptController C = SUNAdaptController_H0211(sunctx);
```

*SUNAdaptController* **SUNAdaptController\_H0321**(*SUNContext* sunctx)

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate the  $H_{0321}$  controller from [104], corresponding with the parameters  $k_1 = 1.25$ ,  $k_2 = 0.5$ ,  $k_3 = -0.75$ ,  $k_4 = 0.25$ , and  $k_5 = 0.75$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Added in version 7.3.0.

Usage:

```
SUNAdaptController C = SUNAdaptController_H0321(sunctx);
```

*SUNAdaptController* **SUNAdaptController\_H211**(*SUNContext* sunctx)

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate the  $H_{211}$  controller from [104], corresponding with the parameters  $k_1 = 0.25$ ,  $k_2 = 0.25$ ,  $k_4 = -0.25$ , and  $k_3 = k_5 = 0$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Added in version 7.3.0.

Usage:

```
SUNAdaptController C = SUNAdaptController_H211(sunctx);
```

***SUNAdaptController* SUNAdaptController\_H312(*SUNContext* sunctx)**

This constructor creates and allocates memory for a *SUNAdaptController\_Soderlind* object, set up to replicate the *H312* controller from [104], corresponding with the parameters  $k_1 = 0.125$ ,  $k_2 = 0.25$ ,  $k_3 = 0.125$ ,  $k_4 = -0.375$ , and  $k_5 = -0.125$ .

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Added in version 7.3.0.

Usage:

```
SUNAdaptController C = SUNAdaptController_H312(sunctx);
```

### 13.3 The *SUNAdaptController\_ImExGus* Module

The ImEx Gustafsson implementation of the *SUNAdaptController* class, *SUNAdaptController\_ImExGus*, implements a combination of two adaptivity controllers proposed by K. Gustafsson. His “explicit” controller was proposed in [52], is primarily useful with explicit Runge–Kutta methods, and has the form

$$h' = \begin{cases} h_1 \varepsilon_1^{-1/(p+1)}, & \text{on the first step,} \\ h_n \varepsilon_n^{-k_1^E/(p+1)} \left( \frac{\varepsilon_n}{\varepsilon_{n-1}} \right)^{k_2^E/(p+1)}, & \text{on subsequent steps.} \end{cases} \quad (13.1)$$

Similarly, Gustafsson’s “implicit” controller was proposed in [53] with the form

$$h' = \begin{cases} h_1 \varepsilon_1^{-1/(p+1)}, & \text{on the first step,} \\ h_n \left( \frac{h_n}{h_{n-1}} \right) \varepsilon_n^{-k_1^I/(p+1)} \left( \frac{\varepsilon_n}{\varepsilon_{n-1}} \right)^{-k_2^I/(p+1)}, & \text{on subsequent steps.} \end{cases} \quad (13.2)$$

In the above formulas, the default values of  $k_1^E$ ,  $k_2^E$ ,  $k_1^I$ , and  $k_2^I$  are 0.367, 0.268, 0.98, and 0.95, respectively, and  $p$  is the global order of the time integration method.

The *SUNAdaptController\_ImExGus* controller implements both formulas (13.1) and (13.2), and sets its recommended step size as the minimum of these two. It is implemented as a derived *SUNAdaptController* class, and defines its *content* field as:

```
struct _SUNAdaptControllerContent_ImExGus {
    sunrealtype k1e;
    sunrealtype k2e;
    sunrealtype k1i;
    sunrealtype k2i;
    sunrealtype bias;
    sunrealtype ep;
    sunrealtype hp;
    sunbooleantype firststep;
};
```

These entries of the *content* field contain the following information:

- **k1e**, **k2e** - explicit controller parameters used in (13.1).

- **k1i**, **k2i** - implicit controller parameters used in (13.2).
- **bias** - error bias factor, that converts from an input temporal error estimate via  $\varepsilon = \text{bias} * \text{dsm}$ .
- **ep** - storage for the previous error estimate,  $\varepsilon_{n-1}$ .
- **hp** - storage for the previous step size,  $h_{n-1}$ .
- **firststep** - flag indicating whether a step has completed successfully, allowing the formulas above to transition between  $h_1$  and  $h_n$ .

The header file to be included when using this module is `sunadaptcontroller/sunadaptcontroller_imexgus.h`.

The `SUNAdaptController_ImExGus` class provides implementations of all operations relevant to a `SUN_ADAPTCONTROLLER_H` controller listed in §13.1.2. The `SUNAdaptController_ImExGus` class also provides the following additional user-callable routines:

*SUNAdaptController* **SUNAdaptController\_ImExGus**(*SUNContext* sunctx)

This constructor creates and allocates memory for a `SUNAdaptController_ImExGus` object, and inserts its default parameters.

#### Parameters

- **sunctx** – the current *SUNContext* object.

#### Returns

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

Usage:

```
SUNAdaptController C = SUNAdaptController_ImExGus(sunctx);
```

*SUNErrCode* **SUNAdaptController\_SetParams\_ImExGus**(*SUNAdaptController* C, *sunrealtype* k1e, *sunrealtype* k2e, *sunrealtype* k1i, *sunrealtype* k2i)

This user-callable function provides control over the relevant parameters above. This should be called *before* the time integrator is called to evolve the problem.

#### Parameters

- **C** – the `SUNAdaptController_ImExGus` object.
- **k1e** – parameter used within the controller time step estimate.
- **k2e** – parameter used within the controller time step estimate.
- **k1i** – parameter used within the controller time step estimate.
- **k2i** – parameter used within the controller time step estimate.

#### Returns

*SUNErrCode* indicating success or failure.

Usage:

```
retval = SUNAdaptController_SetParams_ImExGus(C, 0.4, 0.3, -1.0, 1.0);
```

#### Note

This routine will be called by *SUNAdaptController\_SetOptions()* when using the key “Cid.params\_imexgus”.



## 13.4 The SUNAdaptController\_MRIHTol Module

Added in version 7.2.0.

### 13.4.1 Mathematical motivation

The MRIHTol implementation of the SUNAdaptController class, `SUNAdaptController_MRIHTol`, implements a general structure for telescopic multirate temporal control. A `SUNAdaptController_MRIHTol` object is constructed using two single-rate controller objects, *HControl* and *TolControl*. The MRIHTol controller assumes that overall solution error at a given time scale results from two types of error:

1. “Slow” temporal errors introduced at the current time scale,

$$\varepsilon_n^S = C(t_n) (h_n^S)^{P+1}, \quad (13.3)$$

where  $C(t)$  is independent of the current time scale step size  $h_n^S$  but may vary in time.

2. “Fast” errors introduced through calls to the next-fastest (“inner”) solver,  $\varepsilon_n^F$ . If this inner solver is called to evolve IVPs over time intervals  $[t_{0,i}, t_{F,i}]$  with a relative tolerance  $\text{RTOL}_n^F$ , then it will result in accumulated errors over these intervals of the form

$$\varepsilon_n^F = c(t_n) h_n^S (\text{RTOL}_n^F),$$

where  $c(t)$  is independent of the tolerance or subinterval width but may vary in time, or equivalently,

$$\varepsilon_n^F = \kappa(t_n) (\text{tolfac}_n^F), \quad (13.4)$$

where  $\text{RTOL}_n^F = \text{RTOL}^S \text{tolfac}_n^F$ , the relative tolerance that was supplied to the current time scale solver is  $\text{RTOL}^S$ , and  $\kappa(t_n) = c(t_n) h_n^S \text{RTOL}^S$  is independent of the relative tolerance factor,  $\text{tolfac}_n^F$ .

Single-rate controllers are constructed to adapt a single parameter, e.g.,  $\delta$ , under an assumption that solution error  $\varepsilon$  depends asymptotically on this parameter via the form

$$\varepsilon = \mathcal{O}(\delta^{q+1}).$$

Both (13.3) and (13.4) fit this form, with control parameters  $h_n^S$  and  $\text{tolfac}_n^F$ , and “orders”  $P$  and 0, respectively. Thus an MRIHTol controller employs *HControl* to adapt  $h_n^S$  to control the current time scale error  $\varepsilon_n^S$ , and it employs *TolControl* to adapt  $\text{tolfac}_n^F$  to control the accumulated inner solver error  $\varepsilon_n^F$ .

To avoid overly large changes in calls to the inner solver, we apply bounds on the results from *TolControl*. If *TolControl* predicts a control parameter  $\text{tolfac}'$ , we obtain the eventual tolerance factor via enforcing the following bounds:

$$\begin{aligned} \frac{\text{tolfac}_n^F}{\text{tolfac}'} &\leq \text{relch}_{\max}, \\ \frac{\text{tolfac}'}{\text{tolfac}_n^F} &\leq \text{relch}_{\max}, \\ \text{tolfac}_{\min} &\leq \text{tolfac}' \leq \text{tolfac}_{\max}. \end{aligned}$$

The default values for these bounds are  $\text{relch}_{\max} = 20$ ,  $\text{tolfac}_{\min} = 10^{-5}$ , and  $\text{tolfac}_{\max} = 1$ .

### 13.4.2 Implementation

The `SUNAdaptController_MRIHTol` controller is implemented as a derived `SUNAdaptController` class, and its *content* field is defined by the `SUNAdaptControllerContent_MRIHTol_` structure:

struct **SUNAdaptControllerContent\_MRIHTol\_**

The member data structure for an MRIHTol controller

*SUNAdaptController* **HControl**

A single time-scale controller to adapt the current step size,  $h_n^S$ .

*SUNAdaptController* **TolControl**

A single time-scale controller to adapt the inner solver relative tolerance factor,  $\text{reltol}_n^F$ .

*sunrealtype* **inner\_max\_relch**

The parameter  $\text{relch}_{\max}$  above.

*sunrealtype* **inner\_min\_tolfac**

The parameter  $\text{tolfac}_{\min}$  above.

*sunrealtype* **inner\_max\_tolfac**

The parameter  $\text{tolfac}_{\max}$  above.

The header file to be included when using this module is `sunadaptcontroller/sunadaptcontroller_mrihtol.h`.

The `SUNAdaptController_MRIHTol` class provides implementations of all operations relevant to a `SUN_ADAPTCONTROLLER_MRI_H_TOL` controller listed in §13.1.2. This class also provides the following additional user-callable routines:

*SUNAdaptController* **SUNAdaptController\_MRIHTol**(*SUNAdaptController* HControl, *SUNAdaptController* TolControl, *SUNContext* sunctx)

This constructor creates and allocates memory for a `SUNAdaptController_MRIHTol` object, and inserts its default parameters.

#### Parameters

- **HControl** – the slow time step adaptivity controller object.
- **TolControl** – the inner solver tolerance factor adaptivity controller object.
- **sunctx** – the current *SUNContext* object.

#### Returns

if successful, a usable *SUNAdaptController* object; otherwise it will return NULL.

*SUNErrCode* **SUNAdaptController\_SetParams\_MRIHTol**(*SUNAdaptController* C, *sunrealtype* inner\_max\_relch, *sunrealtype* inner\_min\_tolfac, *sunrealtype* inner\_max\_tolfac)

This user-callable function provides control over the relevant parameters above. This should be called *before* the time integrator is called to evolve the problem. If any argument is outside the allowable range, that parameter will be reset to its default value.

#### Parameters

- **C** – the `SUNAdaptController_MRIHTol` object.
- **inner\_max\_relch** – the parameter  $\text{relch}_{\max}$  (must be  $\geq 1$ ).
- **inner\_min\_tolfac** – the parameter  $\text{tolfac}_{\min}$  (must be  $> 0$ ).
- **inner\_max\_tolfac** – the parameter  $\text{tolfac}_{\max}$  (must be  $> 0$ ).

#### Returns

*SUNErrCode* indicating success or failure.

Changed in version 7.5.0: Removed the requirement that `inner_max_tolfac` must be  $\leq 1$

**Note**

This routine will be called by `SUNAdaptController_SetOptions()` when using the key “Cid.params\_-mrihtol”.

**13.4.3 Usage**

Since this adaptivity controller is constructed using multiple single-rate adaptivity controllers, there are a few steps required when setting this up in an application (the steps below in *italics* correspond to the surrounding steps described in the *MRISet usage skeleton*).

1. *Create an inner stepper object to solve the fast (inner) IVP*
2. Configure the inner stepper to use temporal adaptivity. For example, when using an ARKODE inner stepper and the `ARKodeCreateMRISetInnerStepper()` function, then either use its default adaptivity approach or supply a single-rate SUNAdaptController object, e.g.

```
void* inner_arkode_mem = ERKStepCreate(f_f, T0, y, sunctx);
MRISetInnerStepper inner_stepper = nullptr;
retval = ARKodeCreateMRISetInnerStepper(inner_arkode_mem, &inner_stepper);
SUNAdaptController fcontrol = SUNAdaptController_PID(sunctx);
retval = ARKodeSetAdaptController(inner_arkode_mem, fcontrol);
```

3. If using an ARKODE inner stepper, then set the desired temporal error accumulation estimation strategy via a call to `ARKodeSetAccumulatedErrorType()`, e.g.,

```
retval = ARKodeSetAccumulatedErrorType(inner_arkode_mem, ARK_ACCUMERROR_MAX);
```

4. *Create an MRISet object for the slow (outer) integration*
5. Create single-rate controllers for both the slow step size and inner solver tolerance, e.g.,

```
SUNAdaptController scontrol_H = SUNAdaptController_PI(sunctx);
SUNAdaptController scontrol_Tol = SUNAdaptController_I(sunctx);
```

6. Create the multirate controller object, e.g.,

```
SUNAdaptController scontrol = SUNAdaptController_MRIHTol(scontrol_H, scontrol_Tol, sunctx);
```

7. Attach the multirate controller object to MRISet, e.g.,

```
retval = ARKodeSetAdaptController(arkode_mem, scontrol);
```

An example showing the above steps is provided in `examples/arkode/CXX_serial/ark_kpr_nestedmri.cpp`, where multirate controller objects are used for both the slow and intermediate time scales in a 3-time-scale simulation.



# Chapter 14

## Stepper Data Structure

This section presents the *SUNStepper* base class which represents a generic solution procedure for IVPs of the form

$$\dot{v}(t) = f(t, v) + r(t), \quad v(t_0) = v_0, \quad (14.1)$$

on an interval  $t \in [t_0, t_f]$ . The time dependent forcing term,  $r_i(t)$ , is given by

$$r(t) = \sum_{k=0}^{n_{\text{forcing}}-1} \left( \frac{t - t_{\text{shift}}}{t_{\text{scale}}} \right)^k \hat{f}_k. \quad (14.2)$$

*SUNStepper* provides an abstraction over SUNDIALS integrators, custom integrators, exact solution procedures, or other approaches for solving (14.1). These are used, for example, in operator splitting and forcing methods to solve inner IVPs in a flexible way.

### 14.1 The SUNStepper API

Added in version 7.2.0.

As with other SUNDIALS classes, the *SUNStepper* abstract base class is implemented using a C structure containing a content pointer to the derived class member data and a structure of function pointers to the derived class implementations of the virtual methods.

type **SUNStepper**

An object for solving the IVP (14.1).

The actual definition of the *SUNStepper* structure is kept private to allow for the object internals to change without impacting user code. The following sections describe the base class methods and the virtual methods that a must be provided by a derived class.

#### 14.1.1 Base Class Methods

This section describes methods provided by the *SUNStepper* abstract base class that aid the user in implementing derived classes. This includes functions for creating and destroying a generic base class object, attaching and retrieving the derived class content pointer, and setting function pointers to derived class method implementations.

#### 14.1.1.1 Creating and Destroying an Object

In addition to creating an empty *SUNStepper* using *SUNStepper\_Create()* described below, there is the *ARKode-CreateSUNStepper()* function to construct a *SUNStepper* from an ARKODE integrator.

*SUNErrCode* **SUNStepper\_Create**(*SUNContext* suncctx, *SUNStepper* \*stepper)

This function creates a *SUNStepper* object to which a user should attach the member data (content) pointer and method function pointers.

##### Parameters

- **suncctx** – the SUNDIALS simulation context.
- **stepper** – a pointer to a stepper object.

##### Returns

A *SUNErrCode* indicating success or failure.

Example usage:

```
/* create an instance of the base class */
SUNStepper stepper = NULL;
SUNErrCode err = SUNStepper_Create(suncctx, &stepper);
```

##### Note

See §14.1.1.4 and §14.1.1.6 for details on how to attach member data and method function pointers.

*SUNErrCode* **SUNStepper\_Destroy**(*SUNStepper* \*stepper)

This function frees memory allocated by the *SUNStepper* base class and uses the function pointer optionally specified with *SUNStepper\_SetDestroyFn()* to free the content.

##### Parameters

- **stepper** – a pointer to a stepper object.

##### Returns

A *SUNErrCode* indicating success or failure.

##### Note

This function only frees memory allocated within the base class and the base class structure itself. The user is responsible for freeing any memory allocated for the member data (content).

#### 14.1.1.2 Stepping Functions

*SUNErrCode* **SUNStepper\_Evolve**(*SUNStepper* stepper, *sunrealtype* tout, *N\_Vector* vret, *sunrealtype* \*tret)

This function evolves the ODE (14.1) towards the time tout and stores the solution at time tret in vret.

##### Parameters

- **stepper** – the stepper object.
- **tout** – the time to evolve towards.
- **vret** – on output, the state at time tret.

- **tret** – the time corresponding to the output value **vret**.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_OneStep**(*SUNStepper* stepper, *sunrealtype* tout, *N\_Vector* vret, *sunrealtype* \*tret)

This function evolves the ODE (14.1) *one timestep* towards the time tout and stores the solution at time tret in vret.

**Parameters**

- **stepper** – the stepper object.
- **tout** – the time to evolve towards.
- **vret** – on output, the state at time tret.
- **tret** – the time corresponding to the output value vret.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_FullRhs**(*SUNStepper* stepper, *sunrealtype* t, *N\_Vector* v, *N\_Vector* f, *SUNFullRhsMode* mode)

This function computes the full right-hand side function of the ODE,  $f(t, v) + r(t)$  in (14.1) for a given value of the independent variable t and state vector v.

**Parameters**

- **stepper** – the stepper object.
- **t** – the current value of the independent variable.
- **v** – the current value of the dependent variable vector.
- **f** – the output vector for the ODE right-hand side,  $f(t, v) + r(t)$ , in (14.1).
- **mode** – the purpose of the right-hand side evaluation.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_ReInit**(*SUNStepper* stepper, *sunrealtype* t0, *N\_Vector* v0)

This function reinitializes the stepper to solve a new problem with the given initial condition and clears all counters.

**Parameters**

- **stepper** – the stepper object.
- **t0** – the value of the independent variable  $t_0$ .
- **v0** – the value of the dependent variable vector  $v(t_0)$ .

**Returns**

A *SUNErrCode* indicating success or failure.

Added in version 7.3.0.

*SUNErrCode* **SUNStepper\_Reset**(*SUNStepper* stepper, *sunrealtype* tR, *N\_Vector* vR)

This function resets the stepper state to the provided independent variable value and dependent variable vector.

**Parameters**

- **stepper** – the stepper object.

- **tR** – the value of the independent variable  $t_R$ .
- **vR** – the value of the dependent variable vector  $v(t_R)$ .

#### Returns

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_ResetCheckpointIndex**(*SUNStepper* stepper, *suncountertype* ckptIdxR)

This function resets the index at which new checkpoints will be inserted to *ckptIdxR*.

#### Parameters

- **stepper** – the stepper object.
- **ckptIdxR** – the step index to begin checkpointing from

#### Returns

A *SUNErrCode* indicating success or failure.

Added in version 7.3.0.

*SUNErrCode* **SUNStepper\_SetStopTime**(*SUNStepper* stepper, *sunrealtype* tstop)

This function specifies the value of the independent variable  $t$  past which the solution is not to proceed.

#### Parameters

- **stepper** – the stepper object.
- **tstop** – stopping time for the stepper.

#### Returns

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetStepDirection**(*SUNStepper* stepper, *sunrealtype* stepdir)

This function specifies the direction of integration (forward or backward).

#### Parameters

- **stepper** – the stepper object.
- **stepdir** – value whose sign determines the direction. A positive value selects forward integration, a negative value selects backward integration, and zero leaves the current direction unchanged.

#### Returns

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetForcing**(*SUNStepper* stepper, *sunrealtype* tshift, *sunrealtype* tscale, *N\_Vector* \*forcing, int nforcing)

This function sets the data necessary to compute the forcing term (14.2). This includes the shift and scaling factors for the normalized time  $\frac{t-t_{\text{shift}}}{t_{\text{scale}}}$  and the array of polynomial coefficient vectors  $\hat{f}_k$ .

#### Parameters

- **stepper** – a stepper object.
- **tshift** – the time shift to apply to the current time when computing the forcing,  $t_{\text{shift}}$ .
- **tscale** – the time scaling to apply to the current time when computing the forcing,  $t_{\text{scale}}$ .
- **forcing** – a pointer to an array of forcing vectors,  $\hat{f}_k$ .
- **nforcing** – the number of forcing vectors,  $n_{\text{forcing}}$ . A value of 0 effectively eliminates the forcing term.



**Returns**

A *SUNErrCode* indicating success or failure.

**Note**

When integrating the ODE (14.1) the *SUNStepper* is responsible for evaluating ODE right-hand side function  $f(t, v)$  as well as computing and applying the forcing term (14.2) to obtain the full right-hand side of the ODE (14.1).

*SUNErrCode* **SUNStepper\_GetNumSteps**(*SUNStepper* stepper, *suncountertype* \*nst)

This function gets the number of successful time steps taken by the stepper since it was last initialized.

**Parameters**

- **stepper** – the stepper object.
- **nst** – on output, the number of time steps.

**Returns**

A *SUNErrCode* indicating success or failure.

Added in version 7.3.0.

**14.1.1.3 The Right-Hand Side Evaluation Mode**

enum **SUNFullRhsMode**

A flag indicating the purpose of a right-hand side function evaluation.

enumerator **SUN\_FULLRHS\_START**

Evaluate at the beginning of the simulation.

enumerator **SUN\_FULLRHS\_END**

Evaluate at the end of a successful step.

enumerator **SUN\_FULLRHS\_OTHER**

Evaluate elsewhere, e.g., for dense output.

**14.1.1.4 Attaching and Accessing the Content Pointer**

*SUNErrCode* **SUNStepper\_SetContent**(*SUNStepper* stepper, void \*content)

This function attaches a member data (content) pointer to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **content** – a pointer to the stepper member data.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_GetContent**(*SUNStepper* stepper, void \*\*content)

This function retrieves the member data (content) pointer from a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.

- **content** – a pointer to set to the stepper member data pointer.

**Returns**

A *SUNErrCode* indicating success or failure.

#### 14.1.1.5 Handling Warnings and Errors

An implementation of a *SUNStepper* may have a system of warning and error handling that cannot be encoded as a *SUNErrCode* which is the return type of all *SUNStepper* functions. Therefore, we provide the following function to get and set a separate flag associated with a stepper.

*SUNErrCode* **SUNStepper\_SetLastFlag**(*SUNStepper* stepper, int last\_flag)

This function sets a flag that can be used by *SUNStepper* implementations to indicate warnings or errors that occurred during an operation, e.g., *SUNStepper\_Evolve()*.

**Parameters**

- **stepper** – the stepper object.
- **last\_flag** – the flag value.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_GetLastFlag**(*SUNStepper* stepper, int \*last\_flag)

This function provides the last value of the flag used by the *SUNStepper* implementation to indicate warnings or errors that occurred during an operation, e.g., *SUNStepper\_Evolve()*.

**Parameters**

- **stepper** – the stepper object.
- **last\_flag** – A pointer to where the flag value will be written.

**Returns**

A *SUNErrCode* indicating success or failure.

#### 14.1.1.6 Setting Member Functions

The functions in this section are used to specify how each operation on a *SUNStepper* implementation is performed. Technically, all of these functions are optional to call; the functions that need to be attached are determined by the “consumer” of the *SUNStepper*.

*SUNErrCode* **SUNStepper\_SetEvolveFn**(*SUNStepper* stepper, *SUNStepperEvolveFn* fn)

This function attaches a *SUNStepperEvolveFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperEvolveFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetOneStepFn**(*SUNStepper* stepper, *SUNStepperOneStepFn* fn)

This function attaches a *SUNStepperOneStepFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.

- **fn** – the *SUNStepperOneStepFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetFullRhsFn**(*SUNStepper* stepper, *SUNStepperFullRhsFn* fn)

This function attaches a *SUNStepperFullRhsFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperFullRhsFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetReInitFn**(*SUNStepper* stepper, *SUNStepperResetFn* fn)

This function attaches a *SUNStepperReInitFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperReInitFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

Added in version 7.3.0.

*SUNErrCode* **SUNStepper\_SetResetFn**(*SUNStepper* stepper, *SUNStepperResetFn* fn)

This function attaches a *SUNStepperResetFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperResetFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetResetCheckpointIndexFn**(*SUNStepper* stepper,  
*SUNStepperResetCheckpointIndexFn* fn)

This function attaches a *SUNStepperResetCheckpointIndexFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperResetCheckpointIndexFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

Added in version 7.3.0.

*SUNErrCode* **SUNStepper\_SetStopTimeFn**(*SUNStepper* stepper, *SUNStepperSetStopTimeFn* fn)

This function attaches a *SUNStepperSetStopTimeFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperSetStopTimeFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetStepDirectionFn**(*SUNStepper* stepper, *SUNStepperSetStepDirectionFn* fn)

This function attaches a *SUNStepperSetStepDirectionFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperSetStepDirectionFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetForcingFn**(*SUNStepper* stepper, *SUNStepperSetForcingFn* fn)

This function attaches a *SUNStepperSetForcingFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperSetForcingFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNStepper\_SetGetNumStepsFn**(*SUNStepper* stepper, *SUNStepperGetNumStepsFn* fn)

This function attaches a *SUNStepperGetNumStepsFn* function to a *SUNStepper* object.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperGetNumStepsFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

Added in version 7.3.0.

*SUNErrCode* **SUNStepper\_SetDestroyFn**(*SUNStepper* stepper, *SUNStepperDestroyFn* fn)

This function attaches a *SUNStepperDestroyFn* function to a *SUNStepper*. The provided function is responsible for freeing any memory allocated for the *SUNStepper* content.

**Parameters**

- **stepper** – a stepper object.
- **fn** – the *SUNStepperDestroyFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

### 14.1.2 Implementation Specific Methods

This section describes the virtual methods defined by the *SUNStepper* abstract base class.

```
typedef SUNErrCode (*SUNStepperEvolveFn)(SUNStepper stepper, sunrealtype tout, N_Vector vret, sunrealtype *tret)
```

This type represents a function with the signature of *SUNStepper\_Evolve()*.

```
typedef SUNErrCode (*SUNStepperOneStepFn)(SUNStepper stepper, sunrealtype tout, N_Vector vret, sunrealtype *tret)
```

This type represents a function with the signature of *SUNStepper\_OneStep()*.

```
typedef SUNErrCode (*SUNStepperFullRhsFn)(SUNStepper stepper, sunrealtype t, N_Vector v, N_Vector f, SUNFullRhsMode mode)
```

This type represents a function with the signature of *SUNStepper\_FullRhs()*.

```
typedef SUNErrCode (*SUNStepperResetFn)(SUNStepper stepper, sunrealtype tR, N_Vector vR)
```

This type represents a function with the signature of *SUNStepper\_Reset()*.

```
typedef SUNErrCode (*SUNStepperReInitFn)(SUNStepper stepper, sunrealtype tR, N_Vector vR)
```

This type represents a function with the signature of *SUNStepper\_ReInit()*.

Added in version 7.3.0.

```
typedef SUNErrCode (*SUNStepperResetCheckpointIndexFn)(SUNStepper stepper, suncountertype ckptIdxR)
```

This type represents a function with the signature of *SUNStepper\_ResetCheckpointIndex()*.

Added in version 7.3.0.

```
typedef SUNErrCode (*SUNStepperSetStopTimeFn)(SUNStepper stepper, sunrealtype tstop)
```

This type represents a function with the signature of *SUNStepper\_SetStopTime()*.

```
typedef SUNErrCode (*SUNStepperSetStepDirectionFn)(SUNStepper stepper, sunrealtype stepdir)
```

This type represents a function with the signature of *SUNStepper\_SetStepDirection()*.

```
typedef SUNErrCode (*SUNStepperSetForcingFn)(SUNStepper stepper, sunrealtype tshift, sunrealtype tscale, N_Vector *forcing, int nforcing)
```

This type represents a function with the signature of *SUNStepper\_SetForcing()*.

```
typedef SUNErrCode (*SUNStepperDestroyFn)(SUNStepper stepper)
```

This type represents a function with the signature similar to *SUNStepper\_Destroy()* for freeing the content associated with a *SUNStepper*.

```
typedef SUNErrCode (*SUNStepperGetNumStepsFn)(SUNStepper stepper, suncountertype *nst)
```

This type represents a function with the signature of *SUNStepper\_GetNumSteps()*.

Added in version 7.3.0.

## 14.2 Implementing a SUNStepper

To create a *SUNStepper* implementation:

1. Define the stepper-specific content.

This is typically a user-defined structure in C codes, a user-defined class or structure in C++ codes, or a user-defined module in Fortran codes. This content should hold any data necessary to perform the operations defined by the *SUNStepper* member functions.

2. Define implementations of the required member functions (see §14.1.2).

These are typically user-defined functions in C, member functions of the user-defined structure or class in C++, or functions contained in the user-defined module in Fortran.

Note that all member functions are passed the *SUNStepper* object and the stepper-specific content can, if necessary, be retrieved using *SUNStepper\_GetContent()*. Stepper-specific warnings and errors can be recorded with *SUNStepper\_SetLastFlag()*.

3. In the user code, before creating the outer memory structure that uses the *SUNStepper*, e.g., with *SplittingStepCreate()* or *ForcingStepCreate()*, do the following:
  1. Create a *SUNStepper* object with *SUNStepper\_Create()*.
  2. Attach a pointer to the stepper content to the *SUNStepper* object with *SUNStepper\_SetContent()* if necessary, e.g., when the content is a C structure.
  3. Attach the member function implementations using the functions described in §14.1.1.6.
4. Attach the *SUNStepper* object to the outer memory structure, e.g., with *SplittingStepCreate()* or *ForcingStepCreate()*.

# Chapter 15

## Adjoint Sensitivity Analysis

This section presents the *SUNAdjointStepper* and *SUNAdjointCheckpointScheme* classes that provide a common interface for adjoint sensitivity analysis (ASA) capabilities. Currently it supports *the ASA capabilities in ARKODE*, while the ASA capabilities in *CVODES* and *IDAS* must be used directly.

### 15.1 Introduction to Adjoint Sensitivity Analysis

This section presents the *SUNAdjointStepper* and *SUNAdjointCheckpointScheme* classes. The *SUNAdjointStepper* represents a generic adjoint sensitivity analysis (ASA) procedure to obtain the adjoint sensitivities of an IVP of the form

$$\dot{y}(t) = f(t, y, p), \quad y(t_0) = y_0(p), \quad y \in \mathbb{R}^N, \quad (15.1)$$

where  $p$  is some set of  $N_s$  problem parameters.

#### Note

The API itself does not implement ASA, but it provides a common interface for ASA capabilities implemented in the SUNDIALS packages. Right now it supports *the ASA capabilities in ARKODE*, while the ASA capabilities in *CVODES* and *IDAS* must be used directly.

Suppose we have a functional  $g(t_f, y(t_f), p)$  for which we would like to compute the gradients  $dg(t_f, y(t_f), p)/dy(t_0)$  and/or  $dg(t_f, y(t_f), p)/dp$ . This most often arises in the form of an optimization problem such as

$$\min_{y(t_0), p} g(t_f, y(t_f), p) \quad (15.2)$$

#### Warning

The CVODES documentation uses  $\lambda$  to represent the adjoint variables needed to obtain the gradient  $dG/dp$  where  $G$  is an integral of  $g$ . Our use of  $\lambda$  in the following is akin to the use of  $\mu$  in the CVODES docs.

The adjoint method is one approach to obtaining the gradients that is particularly efficient when there are relatively few functionals and a large number of parameters. While *CVODES* and *IDAS continuous* adjoint methods (differentiate-

then-discretize), ARKODE provides *discrete* adjoint methods (discretize-then-differentiate). For the continuous approach, we derive and solve the adjoint IVP backwards in time

$$\dot{\lambda}(t) = -f_y^*(t, y, p)\lambda, \quad \lambda(t_f) = g_y^*(t_f, y(t_f), p) \quad (15.3)$$

where  $\lambda(t) \in \mathbb{R}^{N_s}$ ,  $f_y \equiv \partial f / \partial y \in \mathbb{R}^{N \times N}$  and  $g_y \equiv \partial g / \partial y \in \mathbb{R}^{N \times N}$ , are the Jacobians with respect to the dependent variable,  $*$  denotes the Hermitian (conjugate) transpose,  $N$  is the size of the original IVP, and  $N_s$  is the number of parameters. When solved with a numerical time integration scheme, the solution to the continuous adjoint IVP is a numerical approximation of the continuous adjoint sensitivities,

$$\lambda(t_n) \approx g_y(t_f, y(t_n), p), \quad \lambda(t_0) \approx g_y(t_f, y(t_0), p). \quad (15.4)$$

The gradients with respect to the parameters can then be obtained as

$$\frac{dg(t_f, y(t_n), p)}{dp} = \lambda^*(t_n) y_p(t_n) + g_p(t_f, y(t_n), p) + \int_{t_n}^{t_f} \lambda^*(t) f_p(t, y(t_n), p) dt, \quad (15.5)$$

where  $y_p(t) \equiv \partial y(t) / \partial p \in \mathbb{R}^{N \times N_s}$ , and  $g_p \equiv \partial g / \partial p \in \mathbb{R}^{N \times N_s}$  and  $f_p \equiv \partial f / \partial p \in \mathbb{R}^{N \times N_s}$  are the Jacobians with respect to the parameters.

For the discrete adjoint approach, we first numerically discretize the original IVP (15.1) using a time integration scheme,  $\varphi$ , so that

$$y_0 = y(t_0), \quad y_n = \varphi(y_{n-k}, \dots, y_{n-1}, p), \quad k = n, \dots, 1. \quad (15.6)$$

For linear multistep methods  $k \geq 1$  and for one step methods  $k = 1$ . Reformulating the optimization problem for the discrete case, we have

$$\min_{y_0, p} g(t_f, y_n, p) \quad (15.7)$$

The gradients of (15.7) can be computed using the transposed chain rule backwards in time to obtain the discrete adjoint variables  $\lambda_n, \lambda_{n-1}, \dots, \lambda_0$  and  $\mu_n, \mu_{n-1}, \dots, \mu_0$ . The discrete adjoint variables represent the gradients of the discrete cost function (15.7) with respect to changes in the discretized IVP (15.6),

$$\frac{dg}{dy_n} = \lambda_n, \quad \frac{dg}{dp} = \mu_n + \lambda_n^* \left( \frac{\partial y_0}{\partial p} \right). \quad (15.8)$$

### 15.1.1 Discrete vs. Continuous Adjoint Method

It is understood that the continuous adjoint method can be problematic in the context of optimization problems because the continuous adjoint method provides an approximation to the gradient of a continuous cost function while the optimizer is expecting the gradient of the discrete cost function. The discrepancy means that the optimizer can fail to due to inconsistent gradients [48, 49]. On the other hand, the discrete adjoint method provides the exact gradient of the discrete cost function allowing the optimizer to fully converge. Consequently, the discrete adjoint method is often preferable in optimization despite its own drawbacks – such as its (relatively) increased memory usage and the possible introduction of unphysical computational modes [102]. This is not to say that the discrete adjoint approach is always the better choice over the continuous adjoint approach in optimization. Computational efficiency and stability of one approach over the other can be both problem and method dependent. Section 8 in the paper [83] discusses the tradeoffs further and provides numerous references that may help inform users in choosing between the discrete and continuous adjoint approaches.

## 15.2 The SUNAdjointStepper Class

Added in version 7.3.0.



### type **SUNAdjointStepper**

The *SUNAdjointStepper* class provides a package-agnostic interface to SUNDIALS ASA capabilities. It currently only supports the discrete ASA capabilities in the ARKODE package, but in the future this support may be expanded.

## 15.2.1 Class Methods

The *SUNAdjointStepper* class has the following methods:

*SUNErrCode* **SUNAdjointStepper\_Create**(*SUNStepper* fwd\_sunstepper, *sunbooleantype* own\_fwd, *SUNStepper* adj\_sunstepper, *sunbooleantype* own\_adj, *suncountertype* final\_step\_idx, *sunrealtype* tf, *N\_Vector* sf, *SUNAdjointCheckpointScheme* checkpoint\_scheme, *SUNContext* sunctx, *SUNAdjointStepper* \*adj\_stepper)

Creates the *SUNAdjointStepper* object needed to solve the adjoint problem.

### Parameters

- **fwd\_sunstepper** – The *SUNStepper* to be used for forward computations of the original ODE.
- **own\_fwd** – Should *fwd\_sunstepper* be owned (and destroyed) by the *SUNAdjointStepper* or not.
- **adj\_sunstepper** – The *SUNStepper* to be used for the backward integration of the adjoint ODE.
- **own\_adj** – Should *adj\_sunstepper* be owned (and destroyed) by the *SUNAdjointStepper* or not.
- **final\_step\_idx** – The index (step number) of the step corresponding to *t\_f* for the forward ODE.
- **tf** – The terminal time for the forward ODE (the initial time for the adjoint ODE).
- **sf** – The terminal condition for the adjoint ODE.
- **checkpoint\_scheme** – The *SUNAdjointCheckpointScheme* object that determines the checkpointing strategy to use. This should be the same object provided to the forward integrator/stepper.
- **sunctx** – The *SUNContext* for the simulation.
- **adj\_stepper** – The *SUNAdjointStepper* to construct (will be NULL on failure).

### Returns

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_ReInit**(*SUNAdjointStepper* self, *sunrealtype* t0, *N\_Vector* y0, *sunrealtype* tf, *N\_Vector* sf)

Reinitializes the adjoint stepper to solve a new problem of the same size.

### Parameters

- **adj\_stepper** – The adjoint solver object.
- **t0** – The new initial time.
- **y0** – The new initial condition.
- **tf** – The time to start integrating the adjoint system from.

- **sf** – The terminal condition vector of sensitivity solutions  $\partial g/\partial y_0$  and  $\partial g/\partial p$ .

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_Evolve**(*SUNAdjointStepper* adj\_stepper, *sunrealtype* tout, *N\_Vector* sens, *sunrealtype* \*tret)

Integrates the adjoint system.

**Parameters**

- **adj\_stepper** – The adjoint solver object.
- **tout** – The time at which the adjoint solution is desired.
- **sens** – The vector of sensitivity solutions  $\partial g/\partial y_0$  and  $\partial g/\partial p$ .
- **tret** – On return, the time reached by the adjoint solver.

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_OneStep**(*SUNAdjointStepper* adj\_stepper, *sunrealtype* tout, *N\_Vector* sens, *sunrealtype* \*tret)

Evolves the adjoint system backwards one step.

**Parameters**

- **adj\_stepper** – The adjoint solver object.
- **tout** – The time at which the adjoint solution is desired.
- **sens** – The vector of sensitivity solutions  $\partial g/\partial y_0$  and  $\partial g/\partial p$ .
- **tret** – On return, the time reached by the adjoint solver.

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_RecomputeFwd**(*SUNAdjointStepper* adj\_stepper, *suncountertype* start\_idx, *sunrealtype* t0, *N\_Vector* y0, *sunrealtype* tf)

Evolves the forward system in time from (start\_idx, t0) to (stop\_idx, tf) with dense checkpointing.

**Parameters**

- **adj\_stepper** – The SUNAdjointStepper object.
- **start\_idx** – the index of the step, w.r.t. the original forward integration, to begin forward integration from.
- **t0** – the initial time, w.r.t. the original forward integration, to start forward integration from.
- **y0** – the initial state, w.r.t. the original forward integration, to start forward integration from.
- **tf** – the final time, w.r.t. the original forward integration, to stop forward integration at.

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_SetUserData**(*SUNAdjointStepper* adj\_stepper, void \*user\_data)

Sets the user data pointer.

**Parameters**

- **adj\_stepper** – The SUNAdjointStepper object.

- **user\_data** – the user data pointer that will be passed back to user-supplied callback functions.

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_GetNumSteps**(*SUNAdjointStepper* adj\_stepper, *suncountertype* \*num\_steps)

Retrieves the number of steps taken by the adjoint stepper.

**Parameters**

- **adj\_stepper** – The SUNAdjointStepper object.
- **num\_steps** – Pointer to store the number of steps.

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_GetNumRecompute**(*SUNAdjointStepper* adj\_stepper, *suncountertype* \*num\_recompute)

Retrieves the number of recomputation steps (in the forward direction) performed by the adjoint stepper.

**Parameters**

- **adj\_stepper** – The SUNAdjointStepper object.
- **num\_recompute** – Pointer to store the number of recomputations.

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointStepper\_PrintAllStats**(*SUNAdjointStepper* adj\_stepper, FILE \*outfile, *SUNOutputFormat* fmt)

Prints the adjoint stepper statistics/counters in a human-readable table format or CSV format.

**Parameters**

- **adj\_stepper** – The SUNAdjointStepper object.
- **outfile** – A file to write the output to.
- **fmt** – the format to write in (*SUN\_OUTPUTFORMAT\_TABLE* or *SUN\_OUTPUTFORMAT\_CSV*).

**Returns**

A *SUNErrCode* indicating failure or success.

## 15.2.2 User-Supplied Functions

typedef int (\***SUNAdjRhsFn**)(*sunrealtype* t, *N\_Vector* y, *N\_Vector* sens, *N\_Vector* sens\_dot, void \*user\_data)

These functions compute the adjoint ODE right-hand side.

For *ARKODE*, this is

$$\begin{aligned}\Lambda &= f_y^*(t, y, p)\lambda, \quad \text{and if the systems has parameters,} \\ \nu &= f_p^*(t, y, p)\lambda.\end{aligned}$$

and corresponds to (2.70) for explicit Runge–Kutta methods.

**Parameters:**

- **t** – the current value of the independent variable.
- **y** – the current value of the forward solution vector.

- **sens** – a *NVECTOR\_MANYVECTOR* object with two subvectors, the first subvector holds  $\lambda$  and the second holds  $\mu$  and is unused in this function.
- **sens\_dot** – a *NVECTOR\_MANYVECTOR* object with two subvectors, the first subvector holds  $\Lambda$  and the second holds  $\nu$ .
- **user\_data** – the *user\_data* pointer that was passed to *SUNAdjointStepper\_SetUserData()*.

**Returns:**

A *SUNAdjRhsFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case the integrator may attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and an error is raised).

**Note**

Allocation of memory for *y* is handled within the integrator.

The vector *sens\_dot* may be uninitialized on input; it is the user's responsibility to fill this entire vector with meaningful values.

## 15.3 The SUNAdjointCheckpointScheme Class

Added in version 7.3.0.

As with other SUNDIALS classes, the *SUNAdjointCheckpointScheme* abstract base class is implemented using a C structure containing a *content* pointer to the derived class member data and a structure of function pointers to the derived class implementations of the virtual methods.

type **SUNAdjointCheckpointScheme**

A class that provides an interface for checkpointing states during forward integration and accessing them as needed during the backwards integration of the adjoint model.

enum **SUNDataIOMode**

enumerator **SUNDATAIOMODE\_INMEM**

The IO mode for data that is stored in addressable random access memory. The location of the memory (e.g., CPU or GPU) is not specified by this mode.

### 15.3.1 Base Class Methods

*SUNErrCode* **SUNAdjointCheckpointScheme\_NewEmpty**(*SUNContext* sunctx, *SUNAdjointCheckpointScheme* \*cs\_ptr)

**Parameters**

- **sunctx** – The SUNDIALS simulation context
- **cs\_ptr** – on output, a pointer to a new *SUNAdjointCheckpointScheme* object

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointCheckpointScheme\_NeedsSaving**(*SUNAdjointCheckpointScheme* self, *suncountertype* step\_num, *suncountertype* stage\_num, *sunrealtype* t, *sunbooleantype* \*yes\_or\_no)

Determines if the (step\_num, stage\_num) should be checkpointed or not.

#### Parameters

- **self** – the *SUNAdjointCheckpointScheme* object
- **step\_num** – the step number of the checkpoint
- **stage\_num** – the stage number of the checkpoint
- **t** – the time of the checkpoint
- **yes\_or\_no** – boolean indicating if the checkpoint should be saved or not

#### Returns

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointCheckpointScheme\_InsertVector**(*SUNAdjointCheckpointScheme* self, *suncountertype* step\_num, *suncountertype* stage\_num, *sunrealtype* t, *N\_Vector* y)

Inserts the vector as the checkpoint for (step\_num, stage\_num).

#### Parameters

- **self** – the *SUNAdjointCheckpointScheme* object
- **step\_num** – the step number of the checkpoint
- **stage\_num** – the stage number of the checkpoint
- **t** – the time of the checkpoint
- **y** – the state vector to checkpoint

#### Returns

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointCheckpointScheme\_LoadVector**(*SUNAdjointCheckpointScheme* self, *suncountertype* step\_num, *suncountertype* stage\_num, *sunrealtype* t, *sunbooleantype* peek, *N\_Vector* \*yout, *sunrealtype* \*tout)

Loads the checkpointed vector for (step\_num, stage\_num).

#### Parameters

- **self** – the *SUNAdjointCheckpointScheme* object
- **step\_num** – the step number of the checkpoint
- **stage\_num** – the stage number of the checkpoint
- **t** – the desired time of the checkpoint
- **peek** – if true, then the checkpoint will be loaded but not deleted regardless of other implementation-specific settings. If false, then the checkpoint may be deleted depending on the implementation.
- **yout** – the loaded state vector
- **tout** – on output, the time of the checkpoint

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointCheckpointScheme\_EnableDense**(*SUNAdjointCheckpointScheme* self, *sunbooleantype* on\_or\_off)

Enables or disables dense checkpointing (checkpointing every step/stage). When dense checkpointing is disabled, the checkpointing interval that was set when the object was created is restored.

**Parameters**

- **self** – the *SUNAdjointCheckpointScheme* object
- **on\_or\_off** – if true, dense checkpointing will be turned on, if false it will be turned off.

**Returns**

A *SUNErrCode* indicating failure or success.

*SUNErrCode* **SUNAdjointCheckpointScheme\_Destroy**(*SUNAdjointCheckpointScheme* \*cs\_ptr)

Destroys (deallocates) the *SUNAdjointCheckpointScheme* object.

**Parameters**

- **cs\_ptr** – pointer to a *SUNAdjointCheckpointScheme* object

**Returns**

A *SUNErrCode* indicating failure or success.

### 15.3.2 Implementation Specific Methods

This section describes the virtual methods defined by the *SUNAdjointCheckpointScheme* abstract base class.

typedef *SUNErrCode* (\***SUNAdjointCheckpointSchemeNeedsSavingFn**)(*SUNAdjointCheckpointScheme* check\_scheme, *suncountertype* step\_num, *suncountertype* stage\_num, *sunrealtype* t, *sunbooleantype* \*yes\_or\_no)

This type represents a function with the signature of *SUNAdjointCheckpointScheme\_NeedsSaving()*.

typedef *SUNErrCode* (\***SUNAdjointCheckpointSchemeInsertVectorFn**)(*SUNAdjointCheckpointScheme* check\_scheme, *suncountertype* step\_num, *suncountertype* stage\_num, *sunrealtype* t, *N\_Vector* y)

This type represents a function with the signature of *SUNAdjointCheckpointScheme\_InsertVector()*.

typedef *SUNErrCode* (\***SUNAdjointCheckpointSchemeLoadVectorFn**)(*SUNAdjointCheckpointScheme* check\_scheme, *suncountertype* step\_num, *suncountertype* stage\_num, *sunrealtype* t, *sunbooleantype* peek, *N\_Vector* \*yout, *sunrealtype* \*tout)

This type represents a function with the signature of *SUNAdjointCheckpointScheme\_LoadVector()*.

typedef *SUNErrCode* (\***SUNAdjointCheckpointSchemeEnableDenseFn**)(*SUNAdjointCheckpointScheme* check\_scheme, *sunbooleantype* on\_or\_off)

This type represents a function with the signature of *SUNAdjointCheckpointScheme\_EnableDense()*.

typedef *SUNErrCode* (\***SUNAdjointCheckpointSchemeDestroyFn**)(*SUNAdjointCheckpointScheme* \*check\_scheme\_ptr)

This type represents a function with the signature of *SUNAdjointCheckpointScheme\_Destroy()*.

### 15.3.3 Setting Content and Member Functions

These functions can be used to set the content pointer or virtual method pointers as needed when implementing the abstract base class.

---

```
SUNErrCode SUNAdjointCheckpointScheme_SetNeedsSavingFn(SUNAdjointCheckpointScheme self, SUNAdjointCheckpointSchemeNeedsSavingFn fn)
```

This function attaches a *SUNAdjointCheckpointSchemeNeedsSavingFn* function to a *SUNAdjointCheckpointScheme* object.

**Parameters**

- **self** – a checkpoint scheme object.
- **fn** – the *SUNAdjointCheckpointSchemeNeedsSavingFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

```
SUNErrCode SUNAdjointCheckpointScheme_SetInsertVectorFn(SUNAdjointCheckpointScheme self, SUNAdjointCheckpointSchemeInsertVectorFn fn)
```

This function attaches a *SUNAdjointCheckpointSchemeInsertVectorFn* function to a *SUNAdjointCheckpointScheme* object.

**Parameters**

- **self** – a checkpoint scheme object.
- **fn** – the *SUNAdjointCheckpointSchemeInsertVectorFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

```
SUNErrCode SUNAdjointCheckpointScheme_SetLoadVectorFn(SUNAdjointCheckpointScheme self, SUNAdjointCheckpointSchemeLoadVectorFn fn)
```

This function attaches a *SUNAdjointCheckpointSchemeLoadVectorFn* function to a *SUNAdjointCheckpointScheme* object.

**Parameters**

- **self** – a checkpoint scheme object.
- **fn** – the *SUNAdjointCheckpointSchemeLoadVectorFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

```
SUNErrCode SUNAdjointCheckpointScheme_SetDestroyFn(SUNAdjointCheckpointScheme self, SUNAdjointCheckpointSchemeDestroyFn fn)
```

This function attaches a *SUNAdjointCheckpointSchemeDestroyFn* function to a *SUNAdjointCheckpointScheme* object.

**Parameters**

- **self** – a checkpoint scheme object.
- **fn** – the *SUNAdjointCheckpointSchemeDestroyFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNAdjointCheckpointScheme\_SetEnableDenseFn**(*SUNAdjointCheckpointScheme* self, *SUNAdjointCheckpointSchemeEnableDenseFn* fn)

This function attaches a *SUNAdjointCheckpointSchemeEnableDenseFn* function to a *SUNAdjointCheckpointScheme* object.

**Parameters**

- **self** – a checkpoint scheme object.
- **fn** – the *SUNAdjointCheckpointSchemeEnableDenseFn* function to attach.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNAdjointCheckpointScheme\_SetContent**(*SUNAdjointCheckpointScheme* self, void \*content)

This function attaches a member data (content) pointer to a *SUNAdjointCheckpointScheme* object.

**Parameters**

- **self** – a checkpoint scheme object.
- **content** – a pointer to the checkpoint scheme member data.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNAdjointCheckpointScheme\_GetContent**(*SUNAdjointCheckpointScheme* self, void \*\*content)

This function retrieves the member data (content) pointer from a *SUNAdjointCheckpointScheme* object.

**Parameters**

- **self** – a checkpoint scheme object.
- **content** – a pointer to set to the checkpoint scheme member data pointer.

**Returns**

A *SUNErrCode* indicating success or failure.

## 15.4 The SUNAdjointCheckpointScheme\_Fixed Module

The *SUNAdjointCheckpointScheme\_Fixed* module implements a scheme where a checkpoint is saved at some fixed interval (in time steps). The module supports checkpointing of time step states only, or time step states with intermediate stage states as well (for multistage methods). When used with a fixed time step size then the number of checkpoints that will be saved is fixed. However, with adaptive time steps the number of checkpoints stored with this scheme is unbounded.

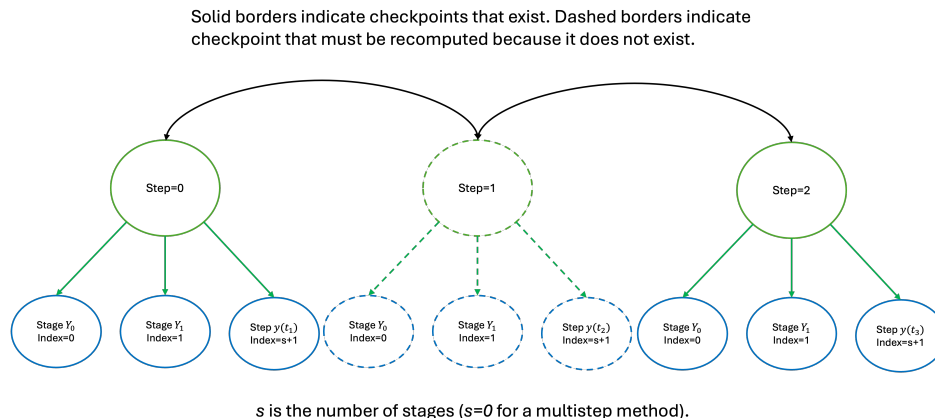
The diagram below illustrates how checkpoints are stored with this scheme:

### 15.4.1 Base-class Method Overrides

The *SUNAdjointCheckpointScheme\_Fixed* module implements the following *SUNAdjointCheckpointScheme* functions:

- *SUNAdjointCheckpointScheme\_NeedsSaving()*
- *SUNAdjointCheckpointScheme\_InsertVector()*
- *SUNAdjointCheckpointScheme\_LoadVector()*





- `SUNAdjointCheckpointScheme_Destroy()`
- `SUNAdjointCheckpointScheme_EnableDense()`

### 15.4.2 Implementation Specific Methods

The `SUNAdjointCheckpointScheme_Fixed` module also implements the following module-specific functions:

`SUNErrCode` `SUNAdjointCheckpointScheme_Create_Fixed`(*SUNDataIOMode* io\_mode, *SUNMemoryHelper* mem\_helper, *suncountertype* interval, *suncountertype* estimate, *sunbooleantype* keep, *SUNContext* sunctx, *SUNAdjointCheckpointScheme* \*check\_scheme\_ptr)

Creates a new *SUNAdjointCheckpointScheme* object that checkpoints at a fixed interval.

#### Parameters

- **io\_mode** – The IO mode used for storing the checkpoints.
- **mem\_helper** – Memory helper for managing memory.
- **interval** – The interval (in steps) between checkpoints.
- **estimate** – An estimate of the total number of checkpoints needed.
- **keep** – Keep data stored even after it is not needed anymore.
- **sunctx** – The *SUNContext* for the simulation.
- **check\_scheme\_ptr** – Pointer to the newly constructed object.

#### Returns

A *SUNErrCode* indicating success or failure.



## Chapter 16

# Tools for Memory Management

To support applications which leverage memory pools, or utilize a memory abstraction layer, SUNDIALS provides a set of utilities that we collectively refer to as the SUNMemoryHelper API. The goal of this API is to allow users to leverage operations defined by native SUNDIALS data structures while allowing the user to have finer-grained control of the memory management.

### 16.1 The SUNMemoryHelper API

This API consists of three new SUNDIALS types: *SUNMemoryType*, *SUNMemory*, and *SUNMemoryHelper*:

typedef struct *SUNMemory\_* \***SUNMemory**

The *SUNMemory* type is a pointer the structure

struct **SUNMemory\_**

void \***ptr**;

The actual data.

*SUNMemoryType* **type**;

The data memory type.

*sunbooleantype* **own**;

A flag indicating ownership.

size\_t **bytes**;

The size of the data allocated.

size\_t **stride**;

Added in version 7.3.0.

The stride of the data.

*SUNMemory* **SUNMemoryNewEmpty**(*SUNContext* suncctx)

This function returns an empty *SUNMemory* object.

#### Parameters

- **suncctx** – the *SUNContext* object.

#### Returns

an uninitialized *SUNMemory* object

Changed in version 7.0.0: The function signature was updated to add the *SUNContext* argument.

enum **SUNMemoryType**

The *SUNMemoryType* type is an enumeration that defines the supported memory types:

enumerator **SUNMEMTYPE\_HOST**

Pageable memory accessible on the host

enumerator **SUNMEMTYPE\_PINNED**

Page-locked memory accessible on the host

enumerator **SUNMEMTYPE\_DEVICE**

Memory accessible from the device

enumerator **SUNMEMTYPE\_UVM**

Memory accessible from the host or device

typedef struct *SUNMemoryHelper\_* **\*SUNMemoryHelper**

The *SUNMemoryHelper* type is a pointer to the structure

struct **SUNMemoryHelper\_**

void **\*content**;

Pointer to the implementation-specific member data

void **\*queue**;

Pointer to the implementation-specific queue (e.g., a *cudaStream\_t\**) to use by default when one is not provided for an operation

Added in version 7.3.0.

*SUNMemoryHelper\_Ops* **ops**;

A virtual method table of member functions

*SUNContext* **sunctx**;

The SUNDIALS simulation context

typedef struct *SUNMemoryHelper\_Ops\_* **\*SUNMemoryHelper\_Ops**

The *SUNMemoryHelper\_Ops* type is defined as a pointer to the structure containing the function pointers to the member function implementations. This structure is define as

struct **SUNMemoryHelper\_Ops\_**

*SUNErrCode* (**\*alloc**)(*SUNMemoryHelper*, *SUNMemory* \*memptr, size\_t mem\_size, *SUNMemoryType* mem\_type, void \*queue)

The function implementing *SUNMemoryHelper\_Alloc()*

*SUNErrCode* (**\*allocstrided**)(*SUNMemoryHelper*, *SUNMemory* \*memptr, size\_t mem\_size, size\_t stride, *SUNMemoryType* mem\_type, void \*queue)

The function implementing *SUNMemoryHelper\_AllocStrided()*

Added in version 7.3.0.

*SUNErrCode* (**\*dealloc**)(*SUNMemoryHelper*, *SUNMemory* mem, void \*queue)

The function implementing *SUNMemoryHelper\_Dealloc()*

*SUNErrCode* (**\*copy**)(*SUNMemoryHelper*, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

The function implementing *SUNMemoryHelper\_Copy()*

*SUNErrCode* (\*copyasync)(*SUNMemoryHelper*, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

The function implementing *SUNMemoryHelper\_CopyAsync()*

*SUNErrCode* (\*getallocstats)(*SUNMemoryHelper*, *SUNMemoryType* mem\_type, unsigned long \*num\_allocations, unsigned long \*num\_deallocations, size\_t \*bytes\_allocated, size\_t \*bytes\_high\_watermark)

The function implementing *SUNMemoryHelper\_GetAllocStats()*

*SUNMemoryHelper* (\*clone)(*SUNMemoryHelper*)

The function implementing *SUNMemoryHelper\_Clone()*

*SUNErrCode* (\*destroy)(*SUNMemoryHelper*)

The function implementing *SUNMemoryHelper\_Destroy()*

### 16.1.1 Implementation defined operations

The *SUNMemory* API defines the following operations that an implementation to must define:

*SUNMemory* **SUNMemoryHelper\_Alloc**(*SUNMemoryHelper* helper, *SUNMemory* \*memptr, size\_t mem\_size, *SUNMemoryType* mem\_type, void \*queue)

Allocates a *SUNMemory* object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

#### Parameters

- **helper** – the *SUNMemoryHelper* object.
- **memptr** – pointer to the allocated *SUNMemory*.
- **mem\_size** – the size in bytes of the ptr.
- **mem\_type** – the *SUNMemoryType* of the ptr.
- **queue** – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

#### Returns

A new *SUNMemory* object

*SUNMemory* **SUNMemoryHelper\_AllocStrided**(*SUNMemoryHelper* helper, *SUNMemory* \*memptr, size\_t mem\_size, size\_t stride, *SUNMemoryType* mem\_type, void \*queue)

Allocates a *SUNMemory* object whose ptr field is allocated for mem\_size bytes with the specified stride, and is of type mem\_type. The new object will have ownership of ptr and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

#### Parameters

- **helper** – the *SUNMemoryHelper* object.
- **memptr** – pointer to the allocated *SUNMemory*.
- **mem\_size** – the size in bytes of the ptr.
- **stride** – the stride of the memory in bytes.
- **mem\_type** – the *SUNMemoryType* of the ptr.
- **queue** – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

**Returns**

A new *SUNMemory* object

Added in version 7.3.0.

*SUNErrCode* **SUNMemoryHelper\_Dealloc**(*SUNMemoryHelper* helper, *SUNMemory* mem, void \*queue)

Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

**Parameters**

- **helper** – the *SUNMemoryHelper* object.
- **mem** – the *SUNMemory* object.
- **queue** – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_Copy**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Synchronously copies mem\_size bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object should use the memory types of dst and src to determine the appropriate transfer type necessary.

**Parameters**

- **helper** – the *SUNMemoryHelper* object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

**Returns**

A *SUNErrCode* indicating success or failure.

## 16.1.2 Utility Functions

The SUNMemoryHelper API defines the following functions which do not require a SUNMemoryHelper instance:

*SUNMemory* **SUNMemoryHelper\_Alias**(*SUNMemoryHelper* helper, *SUNMemory* mem1)

Returns a *SUNMemory* object whose ptr field points to the same address as mem1. The new object *will not* have ownership of ptr, therefore, it will not free ptr when *SUNMemoryHelper\_Dealloc()* is called.

**Parameters**

- **helper** – a *SUNMemoryHelper* object.
- **mem1** – a *SUNMemory* object.

**Returns**

A *SUNMemory* object or NULL if an error occurs.

Changed in version 7.0.0: The *SUNMemoryHelper* argument was added to the function signature.

*SUNMemory* **SUNMemoryHelper\_Wrap**(*SUNMemoryHelper* helper, void \*ptr, *SUNMemoryType* mem\_type)

Returns a *SUNMemory* object whose ptr field points to the ptr argument passed to the function. The new object will not have ownership of ptr, therefore, it will not free ptr when *SUNMemoryHelper\_Dealloc*() is called.

#### Parameters

- **helper** – a *SUNMemoryHelper* object.
- **ptr** – the data pointer to wrap in a *SUNMemory* object.
- **mem\_type** – the *SUNMemoryType* of the ptr.

#### Returns

A *SUNMemory* object or NULL if an error occurs.

Changed in version 7.0.0: The *SUNMemoryHelper* argument was added to the function signature.

*SUNMemoryHelper* **SUNMemoryHelper\_NewEmpty**(*SUNContext* sunctx)

Returns an empty *SUNMemoryHelper*. This is useful for building custom *SUNMemoryHelper* implementations.

#### Parameters

- **helper** – a *SUNMemoryHelper* object.

#### Returns

A *SUNMemoryHelper* object or NULL if an error occurs.

Changed in version 7.0.0: The *SUNMemoryHelper* argument was added to the function signature.

*SUNErrCode* **SUNMemoryHelper\_CopyOps**(*SUNMemoryHelper* src, *SUNMemoryHelper* dst)

Copies the ops field of src to the ops field of dst. This is useful for building custom *SUNMemoryHelper* implementations.

#### Parameters

- **src** – the object to copy from.
- **dst** – the object to copy to.

#### Returns

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_GetAllocStats**(*SUNMemoryHelper* helper, *SUNMemoryType* mem\_type, unsigned long \*num\_allocations, unsigned long \*num\_deallocations, size\_t \*bytes\_allocated, size\_t \*bytes\_high\_watermark)

Returns statistics about the allocations performed with the helper.

#### Parameters

- **helper** – the *SUNMemoryHelper* object.
- **mem\_type** – the *SUNMemoryType* to get stats for.
- **num\_allocations** – (output argument) number of allocations done through the helper.
- **num\_deallocations** – (output argument) number of deallocations done through the helper.
- **bytes\_allocated** – (output argument) total number of bytes allocated through the helper at the moment this function is called.
- **bytes\_high\_watermark** – (output argument) max number of bytes allocated through the helper at any moment in the lifetime of the helper.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_SetDefaultQueue**(*SUNMemoryHelper* helper, void \*queue)

Sets the default queue for the helper.

**Parameters**

- **helper** – the *SUNMemoryHelper* object.
- **queue** – pointer to the queue to use by default.

**Returns**

A *SUNErrCode* indicating success or failure.

Added in version 7.3.0.

### 16.1.3 Implementation overridable operations with defaults

In addition, the *SUNMemoryHelper* API defines the following *optionally overridable* operations which an implementation may define:

*SUNErrCode* **SUNMemoryHelper\_CopyAsync**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object should use the memory types of `dst` and `src` to determine the appropriate transfer type necessary. The `ctx` argument is used when a different execution stream needs to be provided to perform the copy in, e.g. with CUDA this would be a `cudaStream_t`.

**Parameters**

- **helper** – the *SUNMemoryHelper* object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

An int flag indicating success (zero) or failure (non-zero).

**Note**

If this operation is not defined by the implementation, then *SUNMemoryHelper\_Copy()* will be used.

*SUNMemoryHelper* **SUNMemoryHelper\_Clone**(*SUNMemoryHelper* helper)

Clones the *SUNMemoryHelper* object itself.

**Parameters**

- **helper** – the *SUNMemoryHelper* object to clone.

**Returns**

A *SUNMemoryHelper* object.



**Note**

If this operation is not defined by the implementation, then the default clone will only copy the `SUNMemoryHelper_Ops` structure stored in `helper->ops`, and not the `helper->content` field.

*SUNErrCode* **SUNMemoryHelper\_Destroy**(*SUNMemoryHelper* helper)

Destroys (frees) the *SUNMemoryHelper* object itself.

**Parameters**

- **helper** – the *SUNMemoryHelper* object to destroy.

**Returns**

A *SUNErrCode* indicating success or failure.

**Note**

If this operation is not defined by the implementation, then the default destroy will only free the `helper->ops` field and the `helper` itself. The `helper->content` field will not be freed.

### 16.1.4 Implementing a custom SUNMemoryHelper

A particular implementation of the *SUNMemoryHelper* API must:

- Define and implement the required operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one *SUNMemoryHelper* module in the same code.
- Optionally, specify the *content* field of *SUNMemoryHelper*.
- Optionally, define and implement additional user-callable routines acting on the newly defined *SUNMemoryHelper*.

An example of a custom *SUNMemoryHelper* is given in `examples/utilities/custom_memory_helper.h`.

## 16.2 The SUNMemoryHelper\_Sys Implementation

The *SUNMemoryHelper\_Sys* module is an implementation of the *SUNMemoryHelper* API that interfaces with standard library memory management through `malloc/free`. The implementation defines the constructor

*SUNMemoryHelper* **SUNMemoryHelper\_Sys**(*SUNContext* suncctx)

Allocates and returns a *SUNMemoryHelper* object for handling system memory if successful. Otherwise, it returns `NULL`.

### 16.2.1 SUNMemoryHelper\_Sys API Functions

The implementation provides the following operations defined by the *SUNMemoryHelper* API:

- *SUNMemoryHelper\_Alloc*()
- *SUNMemoryHelper\_AllocStrided*()
- *SUNMemoryHelper\_Dealloc*()
- *SUNMemoryHelper\_Copy*()

- *SUNMemoryHelper\_Clone()*
- *SUNMemoryHelper\_GetAllocStats()*
- *SUNMemoryHelper\_Destroy()*

**Note**

The *SUNMemoryHelper\_Sys* always supports *SUNMEMTYPE\_HOST*. If your system also supports allocating unified/coherent memory between CPU and GPU device with *malloc*, then *SUNMEMTYPE\_UVM* is also supported.

## 16.3 The *SUNMemoryHelper\_Cuda* Implementation

The *SUNMemoryHelper\_Cuda* module is an implementation of the *SUNMemoryHelper* API that interfaces to the NVIDIA [5] library. The implementation defines the constructor

*SUNMemoryHelper* ***SUNMemoryHelper\_Cuda***(*SUNContext* *sunctx*)

Allocates and returns a *SUNMemoryHelper* object for handling CUDA memory if successful. Otherwise it returns NULL.

**Parameters**

- ***sunctx*** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNMemoryHelper* object; otherwise it will return NULL.

### 16.3.1 *SUNMemoryHelper\_Cuda* API Functions

The implementation provides the following operations defined by the *SUNMemoryHelper* API:

*SUNMemory* ***SUNMemoryHelper\_Alloc\_Cuda***(*SUNMemoryHelper* *helper*, *SUNMemory* *memptr*, *size\_t* *mem\_size*, *SUNMemoryType* *mem\_type*, void \**queue*)

Allocates a *SUNMemory* object whose *ptr* field is allocated for *mem\_size* bytes and is of type *mem\_type*. The new object will have ownership of *ptr* and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

**Parameters**

- ***helper*** – the *SUNMemoryHelper* object.
- ***memptr*** – pointer to the allocated *SUNMemory*.
- ***mem\_size*** – the size in bytes of the *ptr*.
- ***mem\_type*** – the *SUNMemoryType* of the *ptr*. Supported values are: \* *SUNMEMTYPE\_HOST* – memory is allocated with a call to *malloc*. \* *SUNMEMTYPE\_PINNED* – memory is allocated with a call to *cudaMallocHost*. \* *SUNMEMTYPE\_DEVICE* – memory is allocated with a call to *cudaMalloc*. \* *SUNMEMTYPE\_UVM* – memory is allocated with a call to *cudaMallocManaged*.
- ***queue*** – currently unused.

**Returns**

A new *SUNMemory* object.

*SUNMemory* **SUNMemoryHelper\_AllocStrided\_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size\_t mem\_size, size\_t stride, *SUNMemoryType* mem\_type, void \*queue)

Allocates a *SUNMemory* object whose *ptr* field is allocated for *mem\_size* bytes and is of type *mem\_type*. The new object will have ownership of *ptr* and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

#### Parameters

- **helper** – the *SUNMemoryHelper* object.
- **memptr** – pointer to the allocated *SUNMemory*.
- **mem\_size** – the size in bytes of the *ptr*.
- **stride** – the number of bytes between elements in the array.
- **mem\_type** – the *SUNMemoryType* of the *ptr*.
- **queue** – currently unused.

#### Returns

A new *SUNMemory* object.

Added in version 7.3.0.

*SUNErrCode* **SUNMemoryHelper\_Dealloc\_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* mem, void \*queue)

Deallocates the *mem*→*ptr* field if it is owned by *mem*, and then deallocates the *mem* object.

#### Parameters

- **helper** – the *SUNMemoryHelper* object.
- **mem** – the *SUNMemory* object.
- **queue** – currently unused.

#### Returns

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_Copy\_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Synchronously copies *mem\_size* bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The *helper* object will use the memory types of *dst* and *src* to determine the appropriate transfer type necessary.

#### Parameters

- **helper** – the *SUNMemoryHelper* object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – currently unused.

#### Returns

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_CopyAsync\_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Asynchronously copies *mem\_size* bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The *helper* object will use the memory types of *dst* and *src* to determine the appropriate transfer type necessary.

**Parameters**

- **helper** – the `SUNMemoryHelper` object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – the `cudaStream_t` handle for the stream that the copy will be performed on.

**Returns**

A `SUNErrCode` indicating success or failure.

`SUNErrCode` `SUNMemoryHelper_GetAllocStats_Cuda`(`SUNMemoryHelper` helper, `SUNMemoryType` mem\_type, unsigned long \*num\_allocations, unsigned long \*num\_deallocations, size\_t \*bytes\_allocated, size\_t \*bytes\_high\_watermark)

Returns statistics about memory allocations performed with the helper.

**Parameters**

- **helper** – the `SUNMemoryHelper` object.
- **mem\_type** – the `SUNMemoryType` to get stats for.
- **num\_allocations** – (output argument) number of memory allocations done through the helper.
- **num\_deallocations** – (output argument) number of memory deallocations done through the helper.
- **bytes\_allocated** – (output argument) total number of bytes allocated through the helper at the moment this function is called.
- **bytes\_high\_watermark** – (output argument) max number of bytes allocated through the helper at any moment in the lifetime of the helper.

**Returns**

A `SUNErrCode` indicating success or failure.

## 16.4 The `SUNMemoryHelper_Hip` Implementation

The `SUNMemoryHelper_Hip` module is an implementation of the `SUNMemoryHelper` API that interfaces to the AMD ROCm HIP library [2]. The implementation defines the constructor

`SUNMemoryHelper` `SUNMemoryHelper_Hip`(`SUNContext` sunctx)

Allocates and returns a `SUNMemoryHelper` object for handling HIP memory if successful. Otherwise it returns `NULL`.

**Parameters**

- **sunctx** – the current `SUNContext` object.

**Returns**

if successful, a usable `SUNMemoryHelper` object; otherwise it will return `NULL`.

### 16.4.1 SUNMemoryHelper\_Hip API Functions

The implementation provides the following operations defined by the SUNMemoryHelper API:

*SUNMemory* **SUNMemoryHelper\_Alloc\_Hip**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size\_t mem\_size, *SUNMemoryType* mem\_type, void \*queue)

Allocates a SUNMemory object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

#### Parameters

- **helper** – the SUNMemoryHelper object.
- **memptr** – pointer to the allocated SUNMemory.
- **mem\_size** – the size in bytes of the ptr.
- **mem\_type** – the SUNMemoryType of the ptr. Supported values are: \* SUNMEMTYPE\_HOST – memory is allocated with a call to malloc. \* SUNMEMTYPE\_PINNED – memory is allocated with a call to hipMallocHost. \* SUNMEMTYPE\_DEVICE – memory is allocated with a call to hipMalloc. \* SUNMEMTYPE\_UVM – memory is allocated with a call to hipMallocManaged.
- **queue** – currently unused.

#### Returns

A new *SUNMemory* object.

*SUNMemory* **SUNMemoryHelper\_AllocStrided\_Hip**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size\_t mem\_size, size\_t stride, *SUNMemoryType* mem\_type, void \*queue)

Allocates a SUNMemory object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

#### Parameters

- **helper** – the SUNMemoryHelper object.
- **memptr** – pointer to the allocated SUNMemory.
- **mem\_size** – the size in bytes of the ptr.
- **stride** – the number of bytes between elements in the array.
- **mem\_type** – the SUNMemoryType of the ptr.
- **queue** – currently unused.

#### Returns

A new *SUNMemory* object.

Added in version 7.3.0.

*SUNErrCode* **SUNMemoryHelper\_Dealloc\_Hip**(*SUNMemoryHelper* helper, *SUNMemory* mem, void \*queue)

Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

#### Parameters

- **helper** – the SUNMemoryHelper object.
- **mem** – the SUNMemory object.
- **queue** – currently unused.

#### Returns

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_Copy\_Hip**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Synchronously copies mem\_size bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst and src to determine the appropriate transfer type necessary.

**Parameters**

- **helper** – the SUNMemoryHelper object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – currently unused.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_CopyAsync\_Hip**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Asynchronously copies mem\_size bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst and src to determine the appropriate transfer type necessary.

**Parameters**

- **helper** – the SUNMemoryHelper object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – the hipStream\_t handle for the stream that the copy will be performed on.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_GetAllocStats\_Hip**(*SUNMemoryHelper* helper, *SUNMemoryType* mem\_type, unsigned long \*num\_allocations, unsigned long \*num\_deallocations, size\_t \*bytes\_allocated, size\_t \*bytes\_high\_watermark)

Returns statistics about memory allocations performed with the helper.

**Parameters**

- **helper** – the SUNMemoryHelper object.
- **mem\_type** – the SUNMemoryType to get stats for.
- **num\_allocations** – (output argument) number of memory allocations done through the helper.
- **num\_deallocations** – (output argument) number of memory deallocations done through the helper.
- **bytes\_allocated** – (output argument) total number of bytes allocated through the helper at the moment this function is called.
- **bytes\_high\_watermark** – (output argument) max number of bytes allocated through the helper at any moment in the lifetime of the helper.

**Returns**

A *SUNErrCode* indicating success or failure.

## 16.5 The SUNMemoryHelper\_Sycl Implementation

The SUNMemoryHelper\_Sycl module is an implementation of the SUNMemoryHelper API that interfaces to the SYCL abstraction layer. The implementation defines the constructor

*SUNMemoryHelper* **SUNMemoryHelper\_Sycl**(*SUNContext* sunctx)

Allocates and returns a SUNMemoryHelper object for handling SYCL memory using the provided queue. Otherwise it returns NULL.

**Parameters**

- **sunctx** – the current *SUNContext* object.

**Returns**

if successful, a usable *SUNMemoryHelper* object; otherwise it will return NULL.

### 16.5.1 SUNMemoryHelper\_Sycl API Functions

The implementation provides the following operations defined by the SUNMemoryHelper API:

*SUNMemory* **SUNMemoryHelper\_Alloc\_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size\_t mem\_size, *SUNMemoryType* mem\_type, void \*queue)

Allocates a SUNMemory object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

**Parameters**

- **helper** – the SUNMemoryHelper object.
- **memptr** – pointer to the allocated SUNMemory.
- **mem\_size** – the size in bytes of the ptr.
- **mem\_type** – the SUNMemoryType of the ptr. Supported values are: \* SUNMEMTYPE\_HOST – memory is allocated with a call to malloc. \* SUNMEMTYPE\_PINNED – memory is allocated with a call to `sycl::malloc_host`. \* SUNMEMTYPE\_DEVICE – memory is allocated with a call to `sycl::malloc_device`. \* SUNMEMTYPE\_UVM – memory is allocated with a call to `sycl::malloc_shared`.
- **queue** – the `sycl::queue` handle for the stream that the allocation will be performed on.

**Returns**

A new *SUNMemory* object.

*SUNMemory* **SUNMemoryHelper\_AllocStrided\_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size\_t mem\_size, size\_t stride, *SUNMemoryType* mem\_type, void \*queue)

Allocates a SUNMemory object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated when *SUNMemoryHelper\_Dealloc()* is called.

**Parameters**

- **helper** – the SUNMemoryHelper object.
- **memptr** – pointer to the allocated SUNMemory.

- **mem\_size** – the size in bytes of the ptr.
- **stride** – the number of bytes between elements in the array.
- **mem\_type** – the `SUNMemoryType` of the ptr.
- **queue** – the `sycl::queue` handle for the stream that the allocation will be performed on.

**Returns**

A new `SUNMemory` object.

Added in version 7.3.0.

*SUNErrCode* **SUNMemoryHelper\_Dealloc\_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* mem, void \*queue)

Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

**Parameters**

- **helper** – the `SUNMemoryHelper` object.
- **mem** – the `SUNMemory` object.
- **queue** – the `sycl::queue` handle for the queue that the deallocation will be performed on.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_Copy\_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Synchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

**Parameters**

- **helper** – the `SUNMemoryHelper` object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – the `sycl::queue` handle for the queue that the copy will be performed on.

**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_CopyAsync\_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size\_t mem\_size, void \*queue)

Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

**Parameters**

- **helper** – the `SUNMemoryHelper` object.
- **dst** – the destination memory to copy to.
- **src** – the source memory to copy from.
- **mem\_size** – the number of bytes to copy.
- **queue** – the `sycl::queue` handle for the queue that the copy will be performed on.



**Returns**

A *SUNErrCode* indicating success or failure.

*SUNErrCode* **SUNMemoryHelper\_GetAllocStats\_Sycl**(*SUNMemoryHelper* helper, *SUNMemoryType* mem\_type, unsigned long \*num\_allocations, unsigned long \*num\_deallocations, size\_t \*bytes\_allocated, size\_t \*bytes\_high\_watermark)

Returns statistics about memory allocations performed with the helper.

**Parameters**

- **helper** – the *SUNMemoryHelper* object.
- **mem\_type** – the *SUNMemoryType* to get stats for.
- **num\_allocations** – (output argument) number of memory allocations done through the helper.
- **num\_deallocations** – (output argument) number of memory deallocations done through the helper.
- **bytes\_allocated** – (output argument) total number of bytes allocated through the helper at the moment this function is called.
- **bytes\_high\_watermark** – (output argument) max number of bytes allocated through the helper at any moment in the lifetime of the helper.

**Returns**

A *SUNErrCode* indicating success or failure.



## Chapter 17

# Installing SUNDIALS

In this chapter we discuss two ways for building and installing SUNDIALS from source. The first is with the [Spack](#) HPC package manager and the second is with [CMake](#).

### 17.1 Installing with Spack

Spack is a package management tool that provides a simple spec syntax to configure and install software on a wide variety of platforms and environments. See the [Getting Started](#) section in the Spack documentation for more information on installing Spack.

Once Spack is setup on your system, the default SUNDIALS configuration can be install with the command

```
spack install sundials
```

Additional options can be enabled through Spack package variants. For information on the available variants visit the [SUNDIALS Spack package](#) web page or use the command

```
spack info sundials
```

### 17.2 Installing with CMake

CMake provides a platform-independent build system capable of generating Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. A GUI front end is also available allowing for an interactive build and installation process.

At a minimum, building SUNDIALS requires CMake version 3.18.0 or higher and a working C compiler. If a compatible version of CMake is not already installed on your system, source files or pre-built binary files can be obtained from the [CMake Download website](#).

When building with CMake, you will need to obtain the SUNDIALS source code. You can get the source files by either cloning the [SUNDIALS GitHub repository](#) with the command

```
git clone https://github.com/LLNL/sundials
```

or by downloading release compressed archives (.tar.gz files) from the [SUNDIALS download website](#). The compressed archives allow for downloading the entire SUNDIALS suite or individual packages. The name of the distribution archive is of the form SOLVER-a.b.c.tar.gz, where SOLVER is one of: sundials, cvode, cvodes, arkode,

ida, idas, or kinsol, and a.b.c represents the version number of the SUNDIALS suite or of the individual package. After downloading the relevant archives, uncompress and expand the sources. For example, if you downloaded `sundials-7.7.0.tar.gz`, running the command

```
tar -zxf sundials-7.7.0.tar.gz
```

will extract the source files under the `sundials-7.7.0` directory.

In the installation steps below we will refer to the following directories:

- `SOLVER_DIR` is the `sundials` directory created when cloning from GitHub or the `SOLVER-a.b.c` directory created after uncompressing the release archive.
- `BUILD_DIR` is the (temporary) directory under which SUNDIALS is built. In-source builds are prohibited; the build directory `BUILD_DIR` can **not** be the same as `SOLVER_DIR` and such an attempt will lead to an error. This prevents “polluting” the source tree, simplifies building with different configurations and/or options, and makes it easy to clean-up all traces of the build by simply removing the build directory.
- `INSTALL_DIR` is the directory under which the SUNDIALS exported header files and libraries will be installed. The installation directory `INSTALL_DIR` can not be the same as the `SOLVER_DIR` directory. Typically, header files are exported under a directory `INSTALL_DIR/include` while libraries are typically installed under `INSTALL_DIR/lib` or `INSTALL_LIB/lib64`, with `INSTALL_DIR` specified at configuration time.

### 17.2.1 Linux/Unix systems

CMake can be used from the command line with the `cmake` command, or from graphical interfaces with the `ccmake` or `cmake-gui` commands. Below we present the installation steps using the command line interface.

Using CMake from the command line is simply a matter of generating the build files for the desired configuration, building, and installing. For example, the following commands will build and install the default configuration:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR  
cd BUILD_DIR  
make  
make install
```

The default configuration will install static and shared libraries for all SUNDIALS packages and install the associated example codes. Additional features can be enabled by specifying more options in the configuration step. For example, to enable MPI add `-D SUNDIALS_ENABLE_MPI=ON` to the `cmake` command above:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D SUNDIALS_ENABLE_MPI=ON
```

See section §17.3 below for a complete list of SUNDIALS configuration options and additional configuration examples.

## 17.2.2 Windows Systems

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. Create a separate BUILD\_DIR
2. Open a Visual Studio Command Prompt and cd to BUILD\_DIR
3. Run `cmake-gui ../SOLVER_DIR`
  - a. Hit Configure
  - b. Check/Uncheck solvers to be built
  - c. Change CMAKE\_INSTALL\_PREFIX to INSTALL\_DIR
  - d. Set other options as desired (see section §17.3)
  - e. Hit Generate
4. Back in the VS Command Window:
  - a. Run `msbuild ALL_BUILD.vcxproj`
  - b. Run `msbuild INSTALL.vcxproj`

The resulting libraries will be in the INSTALL\_DIR.

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.

## 17.2.3 HPC Clusters

This section is a guide for installing SUNDIALS on specific HPC clusters. In general, the procedure is the same as described previously in §17.2.1 for Unix/Linux machines. The main differences are in the modules and environment variables that are specific to different HPC clusters. We aim to keep this section as up to date as possible, but it may lag the latest software updates to each cluster.

### 17.2.3.1 Frontier

[Frontier](#) is an Exascale supercomputer at the Oak Ridge Leadership Computing Facility. If you are new to this system, then we recommend that you review the [Frontier user guide](#).

#### A Standard Installation

Load the modules and set the environment variables needed to build SUNDIALS. This configuration enables both MPI and HIP support for distributed and GPU parallelism. It uses the HIP compiler for C and C++ and the Cray Fortran compiler. Other configurations are possible.

```
# required dependencies
module load PrgEnv-cray-amd/8.5.0
module load craype-accel-amd-gfx90a
module load rocm/5.3.0
module load cmake/3.23.2

# GPU-aware MPI
```

(continues on next page)

(continued from previous page)

```

export MPICH_GPU_SUPPORT_ENABLED=1

# compiler environment hints
export CC=$(which hipcc)
export CXX=$(which hipcc)
export FC=$(which ftn)
export CFLAGS="-I${ROCM_PATH}/include"
export CXXFLAGS="-I${ROCM_PATH}/include -Wno-pass-failed"
export LDFLAGS="-L${ROCM_PATH}/lib -lamdhip64 ${PE_MPICH_GTL_DIR_amd_gfx90a} -lmpi_gtl_hsa"

```

Now we can build SUNDIALS. In general, this is the same procedure described in the previous sections. The following command builds and installs SUNDIALS with MPI, HIP, and the Fortran interface enabled, where <account> is your allocation account on Frontier:

```

cmake \
  -S SOLVER_DIR \
  -B BUILD_DIR \
  -D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
  -D AMDGPU_TARGETS=gfx90a \
  -D SUNDIALS_ENABLE_HIP=ON \
  -D SUNDIALS_ENABLE_MPI=ON \
  -D SUNDIALS_ENABLE_FORTRAN=ON
cd BUILD_DIR
make -j8 install
# Need an allocation to run the tests:
salloc -A <account> -t 10 -N 1 -p batch
make test
make test_install_all

```

## 17.3 Configuration options

All available SUNDIALS CMake options are described in the sections below. The default values for some options (e.g., compiler flags and installation paths) are for a Linux system and are provided as illustration only.

### Note

When using a CMake graphical interface (ccmake or cmake-gui), multiple configuration passes are performed before generating the build files. For options where the default value depends on the value of another option, the initial value is set on the first configuration pass and is not updated automatically if the related option value is changed in subsequent passes. For example, the default value of `SUNDIALS_EXAMPLES_INSTALL_PATH` is `CMAKE_INSTALL_PREFIX/examples`; if the value of `CMAKE_INSTALL_PREFIX` is updated, then `SUNDIALS_EXAMPLES_INSTALL_PATH` will also need to be updated as its value was set using the `CMAKE_INSTALL_PREFIX` default.

### 17.3.1 Build Type

The build type determines the level of compiler optimization, if debug information is included, and if additional error checking code is generated. The provided build types are:

- Debug – no optimization flags, debugging information included, additional error checking enabled

- Release – high optimization flags, no debugging information, no additional error checks
- RelWithDebInfo – high optimization flags, debugging information included, no additional error checks
- MinSizeRel – minimize size flags, no debugging information, no additional error checks

Each build type has a corresponding option for the set of compiler flags that are appended to the user-specified compiler flags. See section §17.3.2 for more information.

#### **CMAKE\_BUILD\_TYPE**

Choose the type of build for single-configuration generators (e.g., Makefiles or Ninja).

Default: RelWithDebInfo

#### **CMAKE\_CONFIGURATION\_TYPES**

Specifies the build types for multi-config generators (e.g. Visual Studio, Xcode, or Ninja Multi-Config) as a semicolon-separated list.

Default: Debug, Release, RelWithDebInfo, and MinSizeRel

### **17.3.2 Compilers and Compiler Flags**

Building SUNDIALS requires a C compiler that supports at least a subset of the C99 standard (specifically those features implemented by Visual Studio 2015).

Additional SUNDIALS features that interface with external C++ libraries or GPU programming models require a C++ compiler (e.g., CUDA, HIP, SYCL, Ginkgo, Trilinos, etc.). The C++ standard required depends on the particular library or programming model used and is noted with the relevant options below. The C++ convenience classes provided by SUNDIALS require C++14 or newer. C++ applications that require an earlier C++ standard should use the SUNDIALS C API.

When enabling the SUNDIALS Fortran interfaces, a Fortran compiler that supports the Fortran 2003 or newer standard is required in order to utilize the ISO\_C\_BINDING module.

#### **17.3.2.1 C Compiler**

##### **CMAKE\_C\_COMPILER**

The full path to the C compiler

Default: CMake will attempt to automatically locate a C compiler on the system (e.g., from the CC environment variable or common installation paths).

##### **CMAKE\_C\_FLAGS**

User-specified flags for the C compiler. The value of this option should be a string with flags separated by spaces.

Default: Initialized by the CFLAGS environment variable.

##### **CMAKE\_C\_FLAGS\_DEBUG**

C compiler flags appended when the [CMAKE\\_BUILD\\_TYPE](#) is Debug

Default: -g

##### **CMAKE\_C\_FLAGS\_RELEASE**

C compiler flags appended when the [CMAKE\\_BUILD\\_TYPE](#) is Release

Default: -O3 -DNDEBUG

#### **CMAKE\_C\_FLAGS\_RELWITHDEBINFO**

C compiler flags appended when the *CMAKE\_BUILD\_TYPE* is RelWithDebInfo

Default: -O2 -g -DNDEBUG

#### **CMAKE\_C\_FLAGS\_MINSIZEREL**

C compiler flags appended when the *CMAKE\_BUILD\_TYPE* is MinSizeRel

Default: -Os -DNDEBUG

#### **CMAKE\_C\_STANDARD**

The C standard used when building SUNDIALS C source files.

Default: 99

Options: 99, 11, 17, or 23

#### **CMAKE\_C\_EXTENSIONS**

Enable compiler specific C extensions.

Default: ON

### **17.3.2.2 C++ Compiler**

#### **CMAKE\_CXX\_COMPILER**

The full path to the C++ compiler

Default: CMake will attempt to automatically locate a C++ compiler on the system (e.g., from the CXX environment variable or common installation paths).

#### **CMAKE\_CXX\_FLAGS**

User-specified flags for the C++ compiler. The value of this option should be a string with flags separated by spaces.

Default: Initialized by the CXXFLAGS environment variable.

#### **CMAKE\_CXX\_FLAGS\_DEBUG**

C++ compiler flags appended when the *CMAKE\_BUILD\_TYPE* is Debug

Default: -g

#### **CMAKE\_CXX\_FLAGS\_RELEASE**

C++ compiler flags appended when the *CMAKE\_BUILD\_TYPE* is Release

Default: -O3 -DNDEBUG

#### **CMAKE\_CXX\_FLAGS\_RELWITHDEBINFO**

C++ compiler flags appended when the *CMAKE\_BUILD\_TYPE* is RelWithDebInfo

Default: -O2 -g -DNDEBUG

#### **CMAKE\_CXX\_FLAGS\_MINSIZEREL**

C++ compiler flags appended when the *CMAKE\_BUILD\_TYPE* is MinSizeRel

Default: -Os -DNDEBUG

#### **CMAKE\_CXX\_STANDARD**

The C++ standard used when building SUNDIALS C++ source files.

Default: 14 or 17 if *SUNDIALS\_ENABLE\_GINKGO* or *SUNDIALS\_ENABLE\_SYCL* are ON

Options: 14, 17, 20, or 23



**CMAKE\_CXX\_EXTENSIONS**

Enable compiler specific C++ extensions.

Default: ON

**17.3.2.3 Fortran Compiler****CMAKE\_Fortran\_COMPILER**

The full path to the Fortran compiler

Default: CMake will attempt to automatically locate a Fortran compiler on the system (e.g., from the FC environment variable or common installation paths).

**CMAKE\_Fortran\_FLAGS**

User-specified flags for the Fortran compiler. The value of this option should be a string with flags separated by spaces.

Default: Initialized by the FFLAGS environment variable.

**CMAKE\_Fortran\_FLAGS\_DEBUG**

Fortran compiler flags appended when the *CMAKE\_BUILD\_TYPE* is Debug

Default: -g

**CMAKE\_Fortran\_FLAGS\_RELEASE**

Fortran compiler flags appended when the *CMAKE\_BUILD\_TYPE* is Release

Default: -O3

**CMAKE\_Fortran\_FLAGS\_RELWITHDEBINFO**

Fortran compiler flags appended when the *CMAKE\_BUILD\_TYPE* is RelWithDebInfo

Default: -O2 -g

**CMAKE\_Fortran\_FLAGS\_MINSIZEREL**

Fortran compiler flags appended when the *CMAKE\_BUILD\_TYPE* is MinSizeRel

Default: -Os

**17.3.3 Install Location**

Use the following options to set where the SUNDIALS headers, library, and CMake configuration files will be installed.

**CMAKE\_INSTALL\_PREFIX**

Install path prefix (INSTALL\_DIR), 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 *CMAKE\_INSTALL\_LIBDIR* of *CMAKE\_INSTALL\_PREFIX*, respectively.

**CMAKE\_INSTALL\_LIBDIR**

The directory under *CMAKE\_INSTALL\_PREFIX* where libraries will be installed

Default: Set based on the system as `lib`, `lib64`, or `lib/<multiarch-tuple>`

**SUNDIALS\_INSTALL\_CMAKEDIR**

The directory under *CMAKE\_INSTALL\_PREFIX* where the SUNDIALS CMake package configuration files will be installed (see section §17.6.1 for more information)

Default: `CMAKE_INSTALL_LIBDIR/cmake/sundials`

### 17.3.4 Shared and Static Libraries

Use the following options to set which types of libraries will be installed. By default both static and shared libraries are installed.

**BUILD\_SHARED\_LIBS**

Build shared libraries

Default: ON

**BUILD\_STATIC\_LIBS**

Build static libraries

Default: ON

### 17.3.5 Index Size

**SUNDIALS\_INDEX\_SIZE**

The integer size (in bits) used for indices in SUNDIALS (e.g., for vector and matrix entries), options are: 32 or 64

Default: 64

**Note**

The build system tries to find an integer type of the appropriate size. Candidate 64-bit integer types are (in order of preference): `int64_t`, `__int64`, `long long`, and `long`. Candidate 32-bit integers are (in order of preference): `int32_t`, `int`, and `long`. The advanced option, *SUNDIALS\_INDEX\_TYPE* can be used to provide a type not listed here.

**SUNDIALS\_INDEX\_TYPE**

The integer type used for SUNDIALS indices. The type size must match the size provided in the *SUNDIALS\_INDEX\_SIZE* option.

Default: Automatically determined based on *SUNDIALS\_INDEX\_SIZE*

Changed in version 3.2.0: In prior versions, this option could be set to `INT64_T` to use 64-bit integers or `INT32_T` to use 32-bit integers. These special values are deprecated and a user will only need to use the *SUNDIALS\_INDEX\_SIZE* option in most cases.

### 17.3.6 Precision

#### **SUNDIALS\_PRECISION**

The floating-point precision used in SUNDIALS packages and class implementations, options are: `single`, `double`, or `extended`

Default: `double`

### 17.3.7 Math Library

#### **SUNDIALS\_MATH\_LIBRARY**

The standard C math library (e.g., `libm`) to link with.

Default: `-lm` on Unix systems, none otherwise

### 17.3.8 SUNDIALS Packages

The following options can be used to enable/disable particular SUNDIALS packages.

#### **SUNDIALS\_ENABLE\_ARKODE**

Enable the ARKODE library

Default: `ON`

Added in version 7.7.0: Replaces the deprecated option `BUILD_ARKODE`

#### **SUNDIALS\_ENABLE\_CVODE**

Enable the CVODE library

Default: `ON`

Added in version 7.7.0: Replaces the deprecated option `BUILD_CVODE`

#### **SUNDIALS\_ENABLE\_CVODES**

Enable the CVODES library

Default: `ON`

Added in version 7.7.0: Replaces the deprecated option `BUILD_CVODES`

#### **SUNDIALS\_ENABLE\_IDA**

Enable the IDA library

Default: `ON`

Added in version 7.7.0: Replaces the deprecated option `BUILD_IDA`

#### **SUNDIALS\_ENABLE\_IDAS**

Enable the IDAS library

Default: `ON`

Added in version 7.7.0: Replaces the deprecated option `BUILD_IDAS`

#### **SUNDIALS\_ENABLE\_KINSOL**

Enable the KINSOL library

Default: `ON`

Added in version 7.7.0: Replaces the deprecated option `BUILD_KINSOL`

### 17.3.9 Example Programs

#### **SUNDIALS\_ENABLE\_C\_EXAMPLES**

Build the SUNDIALS C examples

Default: ON

Added in version 7.7.0: Replaces the deprecated option `EXAMPLES_ENABLE_C`

#### **SUNDIALS\_ENABLE\_CXX\_EXAMPLES**

Build the SUNDIALS C++ examples

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `EXAMPLES_ENABLE_CXX`

#### **SUNDIALS\_ENABLE\_CUDA\_EXAMPLES**

Build the SUNDIALS CUDA examples

Default: ON when [SUNDIALS\\_ENABLE\\_CUDA](#) is ON, otherwise OFF

Added in version 7.7.0: Replaces the deprecated option `EXAMPLES_ENABLE_CUDA`

#### **SUNDIALS\_ENABLE\_FORTRAN\_EXAMPLES**

Build the SUNDIALS Fortran 2003 examples

Default: ON when [SUNDIALS\\_ENABLE\\_FORTRAN](#) is ON, otherwise OFF

Added in version 7.7.0: Replaces the deprecated option `EXAMPLES_ENABLE_F2003`

#### **SUNDIALS\_ENABLE\_EXAMPLES\_INSTALL**

Install example program source files and sample output files. See [SUNDIALS\\_EXAMPLES\\_INSTALL\\_PATH](#) for the install location.

A `CMakeLists.txt` file to build the examples will be automatically generated and installed with the source files. If building on a Unix-like system, a `Makefile` for compiling the installed example programs will be also generated and installed.

Default: ON

Added in version 7.7.0: Replaces the deprecated option `EXAMPLES_INSTALL`

#### **SUNDIALS\_EXAMPLES\_INSTALL\_PATH**

Full path to where example source and output files will be installed

Default: `CMAKE_INSTALL_PREFIX/examples`

Added in version 7.7.0: Replaces the deprecated option `EXAMPLES_INSTALL_PATH`

### 17.3.10 Fortran Interfaces

#### **SUNDIALS\_ENABLE\_FORTRAN**

Enable SUNDIALS Fortran interfaces

Default: OFF

#### **Note**

The Fortran interface are only compatible with double precision (i.e., [SUNDIALS\\_PRECISION](#) must be double).

**Warning**

There is a known issue with MSYS/gfortran and SUNDIALS shared libraries that causes linking the Fortran interfaces to fail when building SUNDIALS. For now the work around is to only build with static libraries when using MSYS with gfortran on Windows.

Added in version 7.7.0: Replaces the deprecated option BUILD\_FORTRAN\_MODULE\_INTERFACE

**17.3.11 Error Checking**

For more information on error handling in SUNDIALS, see [Error Checking](#).

**SUNDIALS\_ENABLE\_ERROR\_CHECKS**

Build SUNDIALS with more extensive checks for unrecoverable errors.

Default: ON when [CMAKE\\_BUILD\\_TYPE](#) is Debug, otherwise OFF

**Warning**

Error checks will impact performance, but can be helpful for debugging.

**17.3.12 Logging**

For more information on logging in SUNDIALS, see [Status and Error Logging](#).

**SUNDIALS\_LOGGING\_LEVEL**

The maximum logging level. The options are:

- 0 – no logging
- 1 – log errors
- 2 – log errors + warnings
- 3 – log errors + warnings + informational output
- 4 – log errors + warnings + informational output + debug output
- 5 – log all of the above and even more (e.g. vector valued variables may be logged)

Default: 2

**Warning**

Logging will impact performance, but can be helpful for debugging or understanding algorithm performance. The higher the logging level, the more output that may be logged, and the more performance may degrade.

Changed in version 7.0.0: Enabling MPI in SUNDIALS enables MPI-aware logging.

### 17.3.13 Monitoring

#### **SUNDIALS\_ENABLE\_MONITORING**

Build SUNDIALS with capabilities for fine-grained monitoring of solver progress and statistics. This is primarily useful for debugging.

Default: OFF

##### **Warning**

Building with monitoring may result in minor performance degradation even if monitoring is not utilized.

Added in version 7.7.0: Replaces the deprecated option SUNDIALS\_BUILD\_WITH\_MONITORING

### 17.3.14 Profiling

For more information on profiling in SUNDIALS, see *Performance Profiling*.

#### **SUNDIALS\_ENABLE\_PROFILING**

Build SUNDIALS with capabilities for fine-grained profiling. This requires POSIX timers, the Windows `profileapi.h` timers, or enabling Caliper with [SUNDIALS\\_ENABLE\\_CALIPER](#).

Default: OFF

##### **Warning**

Profiling will impact performance, and should be enabled judiciously.

Added in version 7.7.0: Replaces the deprecated option SUNDIALS\_BUILD\_WITH\_PROFILING

### 17.3.15 Fused Kernels

#### **SUNDIALS\_ENABLE\_PACKAGE\_FUSED\_KERNELS**

Enable fused kernels in SUNDIALS packages

Default: OFF

Added in version 7.7.0: Replaces the deprecated option SUNDIALS\_BUILD\_PACKAGE\_FUSED\_KERNELS

### 17.3.16 Building with Adiak

[Adiak](#) is a library for recording meta-data about HPC simulations. Adiak is developed by Lawrence Livermore National Laboratory and can be obtained from the [Adiak GitHub repository](#).

#### **SUNDIALS\_ENABLE\_ADIAK**

Enable Adiak support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_ADIAK

**adiak\_DIR**

Path to the root of an Adiak installation

Default: None

**SUNDIALS\_ENABLE\_ADIK\_CHECKS**

Perform Adiak compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option `adiak_WORKS`

**17.3.17 Building with Caliper**

[Caliper](#) is a performance analysis library providing a code instrumentation and performance measurement framework for HPC applications. Caliper is developed by Lawrence Livermore National Laboratory and can be obtained from the [Caliper GitHub repository](#).

When profiling and Caliper are both enabled, SUNDIALS will utilize Caliper for performance profiling.

To enable Caliper support, set the `SUNDIALS_ENABLE_CALIPER` to ON and set `CALIPER_DIR` to the root path of the Caliper installation. For example, the following command will configure SUNDIALS with profiling and Caliper support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_PROFILING=ON \
-D SUNDIALS_ENABLE_CALIPER=ON \
-D CALIPER_DIR=/path/to/caliper/installation
```

**SUNDIALS\_ENABLE\_CALIPER**

Enable Caliper support

Default: OFF

**Note**

Using Caliper requires setting `SUNDIALS_ENABLE_PROFILING` to ON.

Added in version 7.7.0: Replaces the deprecated option `ENABLE_CALIPER`

**CALIPER\_DIR**

Path to the root of a Caliper installation

Default: None

**SUNDIALS\_ENABLE\_CALIPER\_CHECKS**

Perform Caliper compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option `CALIPER_WORKS`

### 17.3.18 Building with CUDA

The NVIDIA [CUDA Toolkit](#) provides a development environment for GPU-accelerated computing with NVIDIA GPUs. The CUDA Toolkit and compatible NVIDIA drivers are available from the [NVIDIA developer website](#). SUNDIALS has been tested with the CUDA toolkit versions 10, 11, and 12.

When CUDA support is enabled, the *CUDA NVector*, the *cuSPARSE SUNMatrix*, and the *cuSPARSE batched QR SUNLinearSolver* will be built (see sections §17.7.3.11, §17.7.4.2, and §17.7.5.2, respectively, for the corresponding header files and libraries). For more information on using SUNDIALS with GPUs, see [Features for GPU Accelerated Computing](#).

To enable CUDA support, set `SUNDIALS_ENABLE_CUDA` to ON. If CUDA is installed in a nonstandard location, you may need to set `CUDA_TOOLKIT_ROOT_DIR` to your CUDA Toolkit installation path. You will also need to set `CMAKE_CUDA_ARCHITECTURES` to the CUDA architecture for your system. For example, the following command will configure SUNDIALS with CUDA support for a system with an Ampere GPU:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D SUNDIALS_ENABLE_CUDA=ON \  
-D CMAKE_CUDA_ARCHITECTURES="80"
```

#### `SUNDIALS_ENABLE_CUDA`

Enable CUDA support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_CUDA`

#### `CUDA_TOOLKIT_ROOT_DIR`

Path to the CUDA Toolkit installation

Default: CMake will attempt to automatically locate an installed CUDA Toolkit

#### `CMAKE_CUDA_ARCHITECTURES`

Specifies the CUDA architecture to compile for i.e., 60 for Pascal, 70 for Volta, 80 for Ampere, 90 for Hopper, etc. See the [GPU compute capability tables](#) on the NVIDIA webpage and the [GPU Compilation](#) section of the CUDA documentation for more information.

Default: Determined automatically by CMake. Users are encouraged to override this value with the architecture for their system as the default varies across compilers and compiler versions.

Changed in version 7.2.0: In prior versions `CMAKE_CUDA_ARCHITECTURES` defaulted to 70.

### 17.3.19 Building with Ginkgo

[Ginkgo](#) is a high-performance linear algebra library with a focus on solving sparse linear systems. It is implemented using modern C++ (you will need at least a C++17 compliant compiler to build it), with GPU kernels implemented in CUDA (for NVIDIA devices), HIP (for AMD devices), and SYCL/DPC++ (for Intel devices and other supported hardware). Ginkgo can be obtained from the [Ginkgo GitHub repository](#). SUNDIALS requires using Ginkgo version 1.9.0 or newer and is regularly tested with the latest versions of Ginkgo, specifically versions 1.9.0 and 1.10.0.

When Ginkgo support is enabled, the *Ginkgo SUNMatrix* and *SUNLinearSolver* as well as the *Ginkgo Batch SUNMatrix* and *SUNLinearSolver* header files will be installed (see sections §17.7.4.4 and §17.7.5.4, respectively, for the corresponding header files). For more information on using SUNDIALS with GPUs, see [Features for GPU Accelerated Computing](#).



To enable Ginkgo support, set `SUNDIALS_ENABLE_GINKGO` to ON and set `Ginkgo_DIR` to the root path of the Ginkgo installation. Additionally, set `SUNDIALS_GINKGO_BACKENDS` to a semicolon-separated list of Ginkgo target architectures/executors. For example, the following command will configure SUNDIALS with Ginkgo support using the reference, OpenMP, and CUDA (targeting Ampere GPUs) backends:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_GINKGO=ON \
-D Ginkgo_DIR=/path/to/ginkgo/installation \
-D SUNDIALS_GINKGO_BACKENDS="REF;OMP;CUDA" \
-D SUNDIALS_ENABLE_CUDA=ON \
-D CMAKE_CUDA_ARCHITECTURES="80" \
-D SUNDIALS_ENABLE_OPENMP=ON
```

#### Note

The SUNDIALS interfaces to Ginkgo are not compatible with extended precision (i.e., when `SUNDIALS_PRECISION` is set to extended).

#### **SUNDIALS\_ENABLE\_GINKGO**

Enable Ginkgo support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_GINKGO`

#### **Ginkgo\_DIR**

Path to the Ginkgo installation

Default: None

#### **SUNDIALS\_GINKGO\_BACKENDS**

Semi-colon separated list of Ginkgo target architectures/executors to build for. Options currently supported are REF (the Ginkgo reference executor), OMP (OpenMP), CUDA, HIP, and SYCL.

Default: "REF;OMP"

Changed in version 7.1.0: The DPCPP option was changed to SYCL to align with Ginkgo's naming convention.

#### **SUNDIALS\_ENABLE\_GINKGO\_CHECKS**

Perform Ginkgo compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option `GINKGO_WORKS`

### 17.3.20 Building with HIP

The [Heterogeneous-compute Interface for Portability \(HIP\)](#) allows developers to create portable applications for AMD and NVIDIA GPUs. HIP can be obtained from the [HIP GitHub repository](#). SUNDIALS has been tested with HIP versions between 5.0.0 to 5.4.3.

When HIP support is enabled, the [HIP NVector](#) will be built (see section §17.7.3.12 for the corresponding header file and library). For more information on using SUNDIALS with GPUs, see [Features for GPU Accelerated Computing](#).

To enable HIP support, set `SUNDIALS_ENABLE_HIP` to ON and set `AMDGPU_TARGETS` to the desired target (e.g., `gfx705`). In addition, set `CMAKE_C_COMPILER` and `CMAKE_CXX_COMPILER` to a HIP compatible compiler e.g., `hipcc`. For example, the following command will configure SUNDIALS with HIP support for a system with an MI250X GPU:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D CMAKE_C_COMPILER=hipcc \  
-D CMAKE_CXX_COMPILER=hipcc \  
-D SUNDIALS_ENABLE_HIP=ON \  
-D AMDGPU_TARGETS="gfx90a"
```

**SUNDIALS\_ENABLE\_HIP**

Enable HIP Support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_HIP`

**AMDGPU\_TARGETS**

Specify which AMD GPUs to target

Default: None

### 17.3.21 Building with *hypre*

*hypre* is a library of high performance preconditioners and solvers featuring multigrid methods for the solution of large, sparse linear systems of equations on massively parallel computers. The library is developed by Lawrence Livermore National Laboratory and is available from the [hypre GitHub repository](#). SUNDIALS is regularly tested with the latest versions of *hypre*, specifically up to version 2.26.0.

When *hypre* support is enabled, the *ParHyP NVector* will be built (see section §17.7.3.9 for the corresponding header file and library).

To enable *hypre* support, set `SUNDIALS_ENABLE_MPI` to ON, set `SUNDIALS_ENABLE_HYPRE` to ON, and set `HYPRE_DIR` to the root path of the *hypre* installation. For example, the following command will configure SUNDIALS with *hypre* support:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D SUNDIALS_ENABLE_MPI=ON \  
-D SUNDIALS_ENABLE_HYPRE=ON \  
-D HYPRE_DIR=/path/to/hypre/installation
```

**Note**

SUNDIALS must be configured so that `SUNDIALS_INDEX_SIZE` is compatible with `HYPRE_BigInt` in the *hypre* installation.

**SUNDIALS\_ENABLE\_HYPRE**

Enable *hypre* support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_HYPRE`

#### **HYPRE\_DIR**

Path to the *hypre* installation

Default: None

#### **SUNDIALS\_ENABLE\_HYPRE\_CHECKS**

Perform *hypre* compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option `HYPRE_WORKS`

### **17.3.22 Building with KLU**

KLU is a software package for the direct solution of sparse nonsymmetric linear systems of equations that arise in circuit simulation and is part of [SuiteSparse](#), a suite of sparse matrix software. The library is developed by Texas A&M University and is available from the [SuiteSparse GitHub repository](#). SUNDIALS is regularly tested with the latest versions of KLU, specifically up to SuiteSparse version 7.7.0.

When KLU support is enabled, the *KLU SUNLinearSolver* will be built (see section §17.7.5.5 for the corresponding header file and library).

To enable KLU support, set `SUNDIALS_ENABLE_KLU` to ON. For SuiteSparse 7.4.0 and newer, set `KLU_ROOT` to the root of the SuiteSparse installation. Alternatively, set `KLU_INCLUDE_DIR` and `KLU_LIBRARY_DIR` to the path to the header and library files, respectively, of the SuiteSparse installation. For example, the following command will configure SUNDIALS with KLU support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_KLU=ON \
-D KLU_ROOT=/path/to/suitesparse/installation
```

#### **SUNDIALS\_ENABLE\_KLU**

Enable KLU support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_KLU`

#### **KLU\_ROOT**

Path to the SuiteSparse installation

Default: None

#### **KLU\_INCLUDE\_DIR**

Path to SuiteSparse header files

Default: None

#### **KLU\_LIBRARY\_DIR**

Path to SuiteSparse installed library files

Default: None

**SUNDIALS\_ENABLE\_KLU\_CHECKS**

Perform KLU compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option KLU\_WORKS

**17.3.23 Building with Kokkos**

[Kokkos](#) is a modern C++ (requires at least C++14) programming model for writing performance portable code for multicore CPU and GPU-based systems including NVIDIA, AMD, and Intel GPUs. Kokkos is developed by Sandia National Laboratory and can be obtained from the [Kokkos GitHub repository](#). The minimum supported version of Kokkos 3.7.00. SUNDIALS is regularly tested with the latest versions of Kokkos, specifically up to version 4.3.01.

When Kokkos support is enabled, the *Kokkos NVector* header file will be installed (see section §17.7.3.16 for the corresponding header file). For more information on using SUNDIALS with GPUs, see *Features for GPU Accelerated Computing*.

To enable Kokkos support, set the `SUNDIALS_ENABLE_KOKKOS` to ON and set `Kokkos_DIR` to root path of the Kokkos installation. For example, the following command will configure SUNDIALS with Kokkos support:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D SUNDIALS_ENABLE_KOKKOS=ON \  
-D Kokkos_DIR=/path/to/kokkos/installation
```

**SUNDIALS\_ENABLE\_KOKKOS**

Enable Kokkos support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_KOKKOS

**Kokkos\_DIR**

Path to the Kokkos installation.

Default: None

**SUNDIALS\_ENABLE\_KOKKOS\_CHECKS**

Perform Kokkos compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option KOKKOS\_WORKS

**17.3.24 Building with KokkosKernels**

The [KokkosKernels](#) library is built on Kokkos and provides common linear algebra computational kernels. KokkosKernels is developed by Sandia National Laboratory and can be obtained from the [KokkosKernels GitHub repository](#). The minimum supported version of KokkosKernels 3.7.00. SUNDIALS is regularly tested with the latest versions of KokkosKernels, specifically up to version 4.3.01.

When KokkosKernels support is enabled, the *KokkosKernels SUNMatrix* and *KokkosKernels SUNLinearSolver* header files will be installed (see sections §17.7.4.5 and §17.7.5.6, respectively, for the corresponding header files). For more information on using SUNDIALS with GPUs, see *Features for GPU Accelerated Computing*.

To enable KokkosKernels support, set `SUNDIALS_ENABLE_KOKKOS` and `SUNDIALS_ENABLE_KOKKOS_KERNELS` to ON and set `Kokkos_DIR` and `KokkosKernels_DIR` to the root paths for the Kokkos and KokkosKernels installations, respectively. For example, the following command will configure SUNDIALS with Kokkos and KokkosKernels support:

```
cmake \
  -S SOLVER_DIR \
  -B BUILD_DIR \
  -D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
  -D SUNDIALS_ENABLE_KOKKOS=ON \
  -D Kokkos_DIR=/path/to/kokkos/installation \
  -D SUNDIALS_ENABLE_KOKKOS_KERNELS=ON \
  -D KokkosKernels_DIR=/path/to/kokkoskernels/installation
```

#### **SUNDIALS\_ENABLE\_KOKKOS\_KERNELS**

Enable KokkosKernels support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_KOKKOS_KERNELS`

#### **KokkosKernels\_DIR**

Path to the KokkosKernels installation.

Default: None

#### **SUNDIALS\_ENABLE\_KOKKOS\_KERNELS\_CHECKS**

Perform KokkosKernels compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option `KOKKOS_KERNELS_WORKS`

### **17.3.25 Building with LAPACK**

The **Linear Algebra PACKage (LAPACK)** library interface defines functions for solving systems of linear equations. Several LAPACK implementations are available e.g., the [Netlib reference implementation](#), the [Intel oneAPI Math Kernel Library](#), or [OpenBLAS](#) (among others). SUNDIALS is regularly tested with the latest versions of OpenBLAS, specifically up to version 0.3.27.

When LAPACK support is enabled, the *LAPACK banded SUNLinearSolver* and *LAPACK dense SUNLinearSolver* will be built (see sections §17.7.5.7 and §17.7.5.8, respectively, for the corresponding header files and libraries). Additionally, the *Arnoldi iteration SUNDomEigEstimator* will be build (see §17.7.10.2).

To enable LAPACK support, set `SUNDIALS_ENABLE_LAPACK` to ON. CMake will attempt to find BLAS and LAPACK installations on the system and set the variables `BLAS_LIBRARIES`, `BLAS_LINKER_FLAGS`, `LAPACK_LIBRARIES`, and `LAPACK_LINKER_FLAGS`. You can set the `LAPACK_ROOT` CMake variable to the path of a desired LAPACK installation, and/or set the option `BLA_VENDOR` to tell CMake to only look for LAPACK from a specified vendor (see the [CMake documentation](#)). If necessary, to explicitly override the LAPACK library to build with, manually set the aforementioned variables to the desired values when configuring the build. For example, this is sometimes needed when using OpenBLAS:

```
cmake \
  -S SOLVER_DIR \
  -B BUILD_DIR \
  -D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
  -D SUNDIALS_ENABLE_LAPACK=ON \
```

(continues on next page)

(continued from previous page)

```
-D BLAS_LIBRARIES=/path/to/lapack/installation/lib/libopenblas.so \
-D LAPACK_LIBRARIES=/path/to/lapack/installation/lib/libopenblas.so
```

**Note**

If a working Fortran compiler is not available to infer the name-mangling scheme for LAPACK functions, the options `SUNDIALS_LAPACK_CASE` and `SUNDIALS_LAPACK_UNDERSCORES` *must* be set to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of SUNDIALS were `lower` and `one`, respectively.

**SUNDIALS\_ENABLE\_LAPACK**

Enable LAPACK support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_LAPACK`

**LAPACK\_ROOT**

Path to the LAPACK installation

Default: None

**BLA\_VENDOR**

The LAPACK vendor to search for.

Default: All vendors

**BLAS\_LIBRARIES**

BLAS libraries

Default: None (CMake will try to find a BLAS installation)

**BLAS\_LINKER\_FLAGS**

BLAS required linker flags

Default: None (CMake will try to determine the necessary flags)

**LAPACK\_LIBRARIES**

LAPACK libraries

Default: None (CMake will try to find a LAPACK installation)

**LAPACK\_LINKER\_FLAGS**

LAPACK required linker flags

Default: None (CMake will try to determine the necessary flags)

**SUNDIALS\_LAPACK\_CASE**

Specify the case to use in the Fortran name-mangling scheme, options are: `lower` or `upper`

Default:

**Note**

The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`lower`) scheme if one can not be determined. If used, `SUNDIALS_LAPACK_UNDERSCORES` must also be set.

**SUNDIALS\_LAPACK\_UNDERSCORES**

Specify the number of underscores to append in the Fortran name-mangling scheme, options are: `none`, `one`, or `two`

Default:

**Note**

The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`one`) scheme if one can not be determined. If used, `SUNDIALS_LAPACK_CASE` must also be set.

**SUNDIALS\_ENABLE\_LAPACK\_CHECKS**

Perform LAPACK compatibility checks

Default: `ON`

Added in version 7.7.0: Replaces the deprecated option `LAPACK_WORKS`

**17.3.26 Building with MAGMA**

The [Matrix Algebra on GPU and Multicore Architectures \(MAGMA\)](#) project provides a dense linear algebra library similar to LAPACK but targeting heterogeneous architectures. The library is developed by the University of Tennessee and is available from the [MAGMA GitHub repository](#). SUNDIALS is regularly tested with the latest versions of MAGMA, specifically up to version 2.8.0.

When MAGMA support is enabled, the *MAGMA dense SUNMatrix* and *MAGMA dense SUNLinearSolver* will be built (see sections §17.7.4.6 and §17.7.5.9, respectively, for the corresponding header files and libraries). For more information on using SUNDIALS with GPUs, see *Features for GPU Accelerated Computing*.

To enable MAGMA support, set `SUNDIALS_ENABLE_MAGMA` to `ON`, `MAGMA_DIR` to the root path of MAGMA installation, and `SUNDIALS_MAGMA_BACKENDS` to the desired MAGMA backend to use. For example, the following command will configure SUNDIALS with MAGMA support with the CUDA backend (targeting Ampere GPUs):

```
cmake \
  -S SOLVER_DIR \
  -B BUILD_DIR \
  -D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
  -D SUNDIALS_ENABLE_MAGMA=ON \
  -D MAGMA_DIR=/path/to/magma/installation \
  -D SUNDIALS_MAGMA_BACKEND="CUDA" \
  -D SUNDIALS_ENABLE_CUDA=ON \
  -D CMAKE_CUDA_ARCHITECTURES="80"
```

**SUNDIALS\_ENABLE\_MAGMA**

Enable MAGMA support

Default: `OFF`

Added in version 7.7.0: Replaces the deprecated option `ENABLE_MAGMA`

**MAGMA\_DIR**

Path to the MAGMA installation

Default: `None`

**SUNDIALS\_MAGMA\_BACKENDS**

Which MAGMA backend to use under the SUNDIALS MAGMA interface: CUDA or HIP

Default: CUDA

**SUNDIALS\_ENABLE\_MAGMA\_CHECKS**

Perform MAGMA compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option MAGMA\_WORKS

### 17.3.27 Building with MPI

The [Message Passing Interface \(MPI\)](#) is a standard for communication on parallel computing systems. Several MPI implementations are available e.g., [OpenMPI](#), [MPICH](#), [MVAPICH](#), [Cray MPICH](#), [Intel MPI](#), or [IBM Spectrum MPI](#) (among others). SUNDIALS is regularly tested with the latest versions of OpenMPI, specifically up to version 5.0.5.

When MPI support is enabled, the *parallel NVector*, *MPI ManyVector NVector*, and *MPI+X NVector* will be built (see sections §17.7.3.3, §17.7.3.4, and §17.7.3.5, respectively, for the corresponding header files and libraries).

**Attention**

Changed in version 7.0.0: When MPI is enabled, all SUNDIALS libraries will include MPI symbols and applications will need to include the path for MPI headers and link against the corresponding MPI library.

To enable MPI support, set `SUNDIALS_ENABLE_MPI` to ON. If CMake is unable to locate an MPI installation, set the relevant `MPI_<language>_COMPILER` options to the desired MPI compilers. For example, the following command will configure SUNDIALS with MPI support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_MPI=ON
```

**SUNDIALS\_ENABLE\_MPI**

Enable MPI support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_MPI

**MPI\_C\_COMPILER**

The MPI C compiler e.g., `mpicc`

Default: CMake will attempt to locate an MPI C compiler

**MPI\_CXX\_COMPILER**

The MPI C++ compiler e.g., `mpicxx`

Default: CMake will attempt to locate an MPI C++ compiler



**Note**

This option is only needed if MPI is enabled (*SUNDIALS\_ENABLE\_MPI* is ON) and C++ examples are enabled (*SUNDIALS\_ENABLE\_CXX\_EXAMPLES* is ON). All SUNDIALS solvers can be used from C++ MPI applications by without setting any additional configuration options other than *SUNDIALS\_ENABLE\_MPI*.

**MPI\_Fortran\_COMPILER**

The MPI Fortran compiler e.g., `mpif90`

Default: CMake will attempt to locate an MPI Fortran compiler

**Note**

This option is only needed if MPI is enabled (*SUNDIALS\_ENABLE\_MPI* is ON) and the Fortran interfaces are enabled (*SUNDIALS\_ENABLE\_FORTRAN* is ON).

**MPIEXEC\_EXECUTABLE**

Specify the executable for running MPI programs e.g., `mpiexec`

Default: CMake will attempt to locate the MPI executable

**MPIEXEC\_PREFLAGS**

Specifies flags that come directly after *MPIEXEC\_EXECUTABLE* and before *MPIEXEC\_NUMPROC\_FLAG* and *MPIEXEC\_MAX\_NUMPROCS*.

Default: None

**MPIEXEC\_POSTFLAGS**

Specifies flags that come after the executable to run but before any other program arguments.

Default: None

**17.3.28 Building with oneMKL**

The Intel [oneAPI Math Kernel Library \(oneMKL\)](#) includes CPU and SYCL/DPC++ interfaces for LAPACK dense linear algebra routines. The SUNDIALS oneMKL interface targets the SYCL/DPC++ routines, to utilize the CPU routine see section §17.3.25. SUNDIALS has been tested with oneMKL version 2021.4.

When oneMKL support is enabled, the *oneMKL dense SUNMatrix* and the *oneMKL dense SUNLinearSolver* will be built (see sections §17.7.4.7 and §17.7.5.10, respectively, for the corresponding header files and libraries). For more information on using SUNDIALS with GPUs, see *Features for GPU Accelerated Computing*.

To enable the SUNDIALS oneMKL interface set *SUNDIALS\_ENABLE\_ONEMKL* to ON and *ONEMKL\_DIR* to the root path of oneMKL installation. For example, the following command will configure SUNDIALS with oneMKL support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_ONEMKL=ON \
-D ONEMKL_DIR=/path/to/onemkl/installation \
```

**SUNDIALS\_ENABLE\_ONEMKL**

Enable oneMKL support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_ONEMKL

**ONEMKL\_DIR**

Path to oneMKL installation.

Default: None

**SUNDIALS\_ONEMKL\_USE\_GETRF\_LOOP**

This advanced debugging option replaces the batched LU factorization with a loop over each system in the batch and a non-batched LU factorization.

Default: OFF

**SUNDIALS\_ONEMKL\_USE\_GETRS\_LOOP**

This advanced debugging option replaces the batched LU solve with a loop over each system in the batch and a non-batched solve.

Default: OFF

**SUNDIALS\_ENABLE\_ONEMKL\_CHECKS**

Perform oneMKL compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option ONEMKL\_WORKS

### 17.3.29 Building with OpenMP

The [OpenMP](#) API defines a directive-based approach for portable parallel programming across architectures.

When OpenMP support is enabled, the *OpenMP NVector* will be built (see section §17.7.3.6 for the corresponding header file and library).

To enable OpenMP support, set the [SUNDIALS\\_ENABLE\\_OPENMP](#) to ON. For example, the following command will configure SUNDIALS with OpenMP support:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D SUNDIALS_ENABLE_OPENMP=ON
```

**SUNDIALS\_ENABLE\_OPENMP**

Enable OpenMP support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_OPENMP

### 17.3.30 Building with OpenMP Device Offloading

The [OpenMP](#) 4.0 specification added support for offloading computations to devices (i.e., GPUs). SUNDIALS requires OpenMP 4.5 for GPU offloading support.

When OpenMP offloading support is enabled, the *OpenMPDEV NVector* will be built (see section §17.7.3.7 for the corresponding header file and library).

To enable OpenMP device offloading support, set the `SUNDIALS_ENABLE_OPENMP_DEVICE` to ON. For example, the following command will configure SUNDIALS with OpenMP device offloading support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_OPENMP_DEVICE=ON
```

#### **SUNDIALS\_ENABLE\_OPENMP\_DEVICE**

Enable OpenMP device offloading support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_OPENMP_DEVICE`

#### **SUNDIALS\_ENABLE\_OPENMP\_DEVICE\_CHECKS**

Perform OpenMP device offloading compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option `OPENMP_DEVICE_WORKS`

### **17.3.31 Building with PETSc**

The [Portable, Extensible Toolkit for Scientific Computation \(PETSc\)](#) is a suite of data structures and routines for simulating applications modeled by partial differential equations. The library is developed by Argonne National Laboratory and is available from the [PETSc GitLab repository](#). SUNDIALS requires PETSc 3.5.0 or newer and is regularly tested with the latest versions of PETSc, specifically up to version 3.21.4.

When PETSc support is enabled, the *PETSc NVector* and *PETSc SNES SUNNonlinearSolver* will be built (see sections §17.7.3.10 and §17.7.6.3, respectively, for the corresponding header files and libraries).

To enable PETSc support, set `SUNDIALS_ENABLE_MPI` to ON, set `SUNDIALS_ENABLE_PETSC` to ON, and set `PETSC_DIR` to the path of the PETSc installation. Alternatively, a user can provide a list of include paths in `PETSC_INCLUDES` and a list of complete paths to the PETSc libraries in `PETSC_LIBRARIES`. For example, the following command will configure SUNDIALS with PETSc support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_MPI=ON \
-D SUNDIALS_ENABLE_PETSC=ON \
-D PETSC_DIR=/path/to/petsc/installation
```

#### **SUNDIALS\_ENABLE\_PETSC**

Enable PETSc support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_PETSC`

**PETSC\_DIR**

Path to PETSc installation

Default: None

**PETSC\_LIBRARIES**

Semi-colon separated list of PETSc link libraries. Unless provided by the user, this is autopopulated based on the PETSc installation found in [PETSC\\_DIR](#).

Default: None

**PETSC\_INCLUDES**

Semi-colon separated list of PETSc include directories. Unless provided by the user, this is autopopulated based on the PETSc installation found in [PETSC\\_DIR](#).

Default: None

**SUNDIALS\_ENABLE\_PETSC\_CHECKS**

Perform PETSc compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option PETSC\_WORKS

### 17.3.32 Building with PThreads

POSIX Threads (PThreads) is an API for shared memory programming defined by the Institute of Electrical and Electronics Engineers (IEEE) standard POSIX.1c.

When PThreads support is enabled, the *PThreads NVector* will be built (see section §17.7.3.8 for the corresponding header file and library).

To enable PThreads support, set [SUNDIALS\\_ENABLE\\_PTHREAD](#) to ON. For example, the following command will configure SUNDIALS with PThreads support:

```
cmake \
  -S SOLVER_DIR \
  -B BUILD_DIR \
  -D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
  -D SUNDIALS_ENABLE_PTHREAD=ON
```

**SUNDIALS\_ENABLE\_PTHREAD**

Enable PThreads support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_PTHREAD

### 17.3.33 Building with RAJA

[RAJA](#) is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from the [RAJA GitHub repository](#). SUNDIALS is regularly tested with the latest versions of RAJA, specifically up to version 2024.02.2.

When RAJA support is enabled, the *RAJA NVector* will be built (see section §17.7.3.13 for the corresponding header files and libraries).

To enable RAJA support, set `SUNDIALS_ENABLE_RAJA` to ON, set `RAJA_DIR` to the path of the RAJA installation, set `SUNDIALS_RAJA_BACKENDS` to the desired backend (CUDA, HIP, or SYCL), and set `SUNDIALS_ENABLE_CUDA`, `SUNDIALS_ENABLE_HIP`, or `SUNDIALS_ENABLE_SYCL` to ON depending on the selected backend. For example, the following command will configure SUNDIALS with RAJA support using the CUDA backend (targeting Ampere GPUs):

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_RAJA=ON \
-D RAJA_DIR=/path/to/raja/installation \
-D SUNDIALS_RAJA_BACKENDS="CUDA" \
-D SUNDIALS_ENABLE_CUDA=ON \
-D CMAKE_CUDA_ARCHITECTURES="80"
```

#### **SUNDIALS\_ENABLE\_RAJA**

Enable RAJA support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_RAJA`

#### **RAJA\_DIR**

Path to the RAJA installation

Default: None

#### **SUNDIALS\_RAJA\_BACKENDS**

If building SUNDIALS with RAJA support, this sets the RAJA backend to target. Values supported are CUDA, HIP, or SYCL.

Default: CUDA

### **17.3.34 Building with SuperLU\_DIST**

`SuperLU_DIST` is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations in a distributed memory setting. The library is developed by Lawrence Berkeley National Laboratory and is available from the [SuperLU\\_DIST GitHub repository](#). SuperLU\_DIST version 7.0.0 or newer is required. SUNDIALS is regularly tested with the latest versions of SuperLU\_DIST, specifically up to version 8.2.1.

When SuperLU\_DIST support is enabled, the *SuperLU\_DIST (SLUNRloc) SUNMatrix* and *SuperLU\_DIST SUNLinearSolver* will be built (see sections §17.7.4.9 and §17.7.5.16 for the corresponding header files and libraries).

To enable SuperLU\_DIST support, set `SUNDIALS_ENABLE_MPI` to ON, set `SUNDIALS_ENABLE_SUPERLUDIST` to ON, and set `SUPERLUDIST_DIR` to the path where SuperLU\_DIST is installed. If SuperLU\_DIST was built with OpenMP enabled, set `SUPERLUDIST_OpenMP` and `SUNDIALS_ENABLE_OPENMP` to ON. For example, the following command will configure SUNDIALS with SuperLU\_DIST support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D SUNDIALS_ENABLE_SUPERLUDIST=ON \
-D SUPERLUDIST_DIR=/path/to/superludist/installation
```

#### **SUNDIALS\_ENABLE\_SUPERLUDIST**

Enable SuperLU\_DIST support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_SUPERLUDIST

#### **SUPERLUDIST\_DIR**

Path to SuperLU\_DIST installation.

Default: None

#### **SUPERLUDIST\_OpenMP**

Enable SUNDIALS support for SuperLU\_DIST built with OpenMP

Default: None

##### **Note**

SuperLU\_DIST must be built with OpenMP support for this option to function. Additionally the environment variable OMP\_NUM\_THREADS must be set to the desired number of threads.

#### **SUPERLUDIST\_INCLUDE\_DIRS**

List of include paths for SuperLU\_DIST (under a typical SuperLU\_DIST install, this is typically the SuperLU\_DIST SRC directory)

Default: None

##### **Note**

This is an advanced option. Prefer to use [\*SUPERLUDIST\\_DIR\*](#).

#### **SUPERLUDIST\_LIBRARIES**

Semi-colon separated list of libraries needed for SuperLU\_DIST

Default: None

##### **Note**

This is an advanced option. Prefer to use [\*SUPERLUDIST\\_DIR\*](#).

#### **SUPERLUDIST\_INCLUDE\_DIR**

Path to SuperLU\_DIST header files (under a typical SuperLU\_DIST install, this is typically the SuperLU\_DIST SRC directory)

Default: None

##### **Note**

This is an advanced option. This option is deprecated. Use [\*SUPERLUDIST\\_INCLUDE\\_DIRS\*](#).

**SUPERLUDIST\_LIBRARY\_DIR**

Path to SuperLU\_DIST installed library files

Default: None

**Note**This option is deprecated. Use [SUPERLUDIST\\_DIR](#).**SUNDIALS\_ENABLE\_SUPERLUDIST\_CHECKS**

Perform SuperLU\_DIST compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option SUPERLUDIST\_WORKS

**17.3.35 Building with SuperLU\_MT**

[SuperLU\\_MT](#) is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on shared memory parallel machines. The library is developed by Lawrence Berkeley National Laboratory and is available from the [SuperLU\\_MT GitHub repository](#). SUNDIALS is regularly tested with the latest versions of SuperLU\_MT, specifically up to version 4.0.1.

When SuperLU\_MT support is enabled, the *SuperLU\_MT SUNLinearSolver* will be built (see section §17.7.5.17 for the corresponding header file and library).

To enable SuperLU\_MT support, set [SUNDIALS\\_ENABLE\\_SUPERLUMT](#) to ON, set [SUPERLUMT\\_INCLUDE\\_DIR](#) and [SUPERLUMT\\_LIBRARY\\_DIR](#) to the location of the header and library files, respectively, of the SuperLU\_MT installation. Depending on the SuperLU\_MT installation, it may also be necessary to set [SUPERLUMT\\_LIBRARIES](#) to a semi-colon separated list of other libraries SuperLU\_MT depends on. For example, if SuperLU\_MT was build with an external blas library, then include the full path to the blas library in this list. Additionally, the variable [SUPERLUMT\\_THREAD\\_TYPE](#) must be set to either Pthread or OpenMP. For example, the following command will configure SUNDIALS with SuperLU\_MT support using PThreads:

```
cmake \
  -S SOLVER_DIR \
  -B BUILD_DIR \
  -D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
  -D SUNDIALS_ENABLE_SUPERLUMT=ON \
  -D SUPERLUMT_INCLUDE_DIR=/path/to/superluml/installation/include/dir \
  -D SUPERLUMT_LIBRARY_DIR=/path/to/superluml/installation/library/dir \
  -D SUPERLUMT_THREAD_TYPE="Pthread"
```

**Warning**

Do not mix thread types when using SUNDIALS packages. For example, if using the OpenMP or PThreads NVector then the SuperLU\_MT installation should use the same threading type.

**SUNDIALS\_ENABLE\_SUPERLUMT**

Enable SuperLU\_MT support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option ENABLE\_SUPERLUMT

**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\_LIBRARIES**

Semi-colon separated list of libraries needed for SuperLU\_MT

Default: None

**SUPERLUMT\_THREAD\_TYPE**

Must be set to Pthread or OpenMP, depending on how SuperLU\_MT was compiled.

Default: Pthread

**SUNDIALS\_ENABLE\_SUPERLUMT\_CHECKS**

Perform SuperLU\_MT compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option SUPERLUMT\_WORKS

### 17.3.36 Building with SYCL

**SYCL** is an abstraction layer for programming heterogeneous parallel computing based on C++17.

When SYCL support is enabled, the *SYCL NVector* will be built (see section §17.7.3.14 for the corresponding header file and library).

To enable SYCL support, set the *SUNDIALS\_ENABLE\_SYCL* to ON. For example, the following command will configure SUNDIALS with SYCL support using Intel compilers:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D CMAKE_C_COMPILER=icx \  
-D CMAKE_CXX_COMPILER=icpx \  
-D CMAKE_CXX_FLAGS="-fsycl" \  
-D SUNDIALS_ENABLE_SYCL=ON
```

**SUNDIALS\_ENABLE\_SYCL**

Enable SYCL support

Default: OFF

**Note**

Building with SYCL enabled requires a compiler that supports a subset of the of SYCL 2020 specification (specifically `sycl/sycl.hpp` must be available).



CMake does not currently support autodetection of SYCL compilers and `CMAKE_CXX_COMPILER` must be set to a valid SYCL compiler. At present the only supported SYCL compilers are the Intel oneAPI compilers i.e., `dpcpp` and `icpx`. When using `icpx` the `-fsycl` flag and any ahead of time compilation flags must be added to `CMAKE_CXX_FLAGS`.

Added in version 7.7.0: Replaces the deprecated option `ENABLE_SYCL`

#### **SUNDIALS\_SYCL\_2020\_UNSUPPORTED**

This advanced option disables the use of *some* features from the SYCL 2020 standard in SUNDIALS libraries and examples. This can be used to work around some cases of incomplete compiler support for SYCL 2020.

Default: OFF

### **17.3.37 Building with Trilinos**

**Trilinos** is a collection of C++ libraries of linear solvers, non-linear solvers, optimization solvers, etc. developed by Sandia National Laboratory and available from the [Trilinos GitHub repository](#). SUNDIALS is regularly tested with the latest versions of Trilinos, specifically up to version 16.0.0.

When Trilinos support is enabled, the *Trilinos Tpetra NVector* will be built (see section §17.7.3.15 for the corresponding header file and library).

To enable Trilinos support, set the `SUNDIALS_ENABLE_TRILINOS` to ON and set `Trilinos_DIR` to root path of the Trilinos installation. For example, the following command will configure SUNDIALS with Trilinos support:

```
cmake \
-S SOLVER_DIR \
-B BUILD_DIR \
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \
-D ENABLE_TRILINOS=ON \
-D TRILINOS_DIR=/path/to/trilinos/installation
```

#### **SUNDIALS\_ENABLE\_TRILINOS**

Enable Trilinos support

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_TRILINOS`

#### **Trilinos\_DIR**

Path to the Trilinos installation

Default: None

### **17.3.38 Building with XBraid**

**XBraid** is parallel-in-time library implementing an optimal-scaling multigrid reduction in time (MGRIT) solver. The library is developed by Lawrence Livermore National Laboratory and is available from the [XBraid GitHub repository](#). SUNDIALS is regularly tested with the latest versions of XBraid, specifically up to version 3.0.0.

To enable XBraid support, set `SUNDIALS_ENABLE_MPI` to ON, set `SUNDIALS_ENABLE_XBRAID` to ON, set `XBRAID_DIR` to the root path of the XBraid installation. For example, the following command will configure SUNDIALS with XBraid support:

```
cmake \  
-S SOLVER_DIR \  
-B BUILD_DIR \  
-D CMAKE_INSTALL_PREFIX=INSTALL_DIR \  
-D SUNDIALS_INDEX_SIZE="32" \  
-D SUNDIALS_ENABLE_MPI=ON \  
-D SUNDIALS_ENABLE_XBRAID=ON \  
-D XBRAID_DIR=/path/to/xbraid/installation
```

#### Note

At this time the XBraid types `braid_Int` and `braid_Real` are hard-coded to `int` and `double` respectively. As such SUNDIALS must be configured with `SUNDIALS_INDEX_SIZE` set to 32 and `SUNDIALS_PRECISION` set to `double`. Additionally, SUNDIALS must be configured with `SUNDIALS_ENABLE_MPI` set to `ON`.

#### SUNDIALS\_ENABLE\_XBRAID

Enable or disable the ARKStep + XBraid interface.

Default: OFF

Added in version 7.7.0: Replaces the deprecated option `ENABLE_XBRAID`

#### XBRAID\_DIR

The root directory of the XBraid installation.

Default: OFF

#### XBRAID\_INCLUDES

Semi-colon separated list of XBraid include directories. Unless provided by the user, this is autopopulated based on the XBraid installation found in `XBRAID_DIR`.

Default: None

#### XBRAID\_LIBRARIES

Semi-colon separated list of XBraid link libraries. Unless provided by the user, this is autopopulated based on the XBraid installation found in `XBRAID_DIR`.

Default: None

#### SUNDIALS\_ENABLE\_XBRAID\_CHECKS

Perform XBraid compatibility checks

Default: ON

Added in version 7.7.0: Replaces the deprecated option `XBRAID_WORKS`

### 17.3.39 Building with xSDK Defaults

The [Extreme-scale Scientific Software Development Kit \(xSDK\)](#) is a community of HPC libraries and applications developing best practices and standards for scientific software.

#### USE\_XSDK\_DEFAULTS

Enable xSDK default configuration settings. This sets the default value for `CMAKE_BUILD_TYPE` to `Debug`, `SUNDIALS_INDEX_SIZE` to 32, and `SUNDIALS_PRECISION` to `double`.

Default: OFF

### 17.3.40 Building with External Addons

SUNDIALS “addons” are community developed code additions for SUNDIALS that can be subsumed by the SUNDIALS build system so that they have full access to all internal SUNDIALS symbols. The intent is for SUNDIALS addons to function as if they are part of the SUNDIALS library, while allowing them to potentially have different licenses (although we encourage BSD-3-Clause still), code style (although we encourage them to follow the SUNDIALS style outlined [here](#)).

#### Warning

SUNDIALS addons are not maintained by the SUNDIALS team and may come with different licenses. Use them at your own risk.

To build with SUNDIALS addons,

1. Clone/copy the addon(s) into `SOLVER_DIR/external/`
2. Copy the `sundials-addon-example` block in the `SOLVER_DIR/external/CMakeLists.txt`, paste it below the example block, and modify the path listed for your own external addon(s).
3. When building SUNDIALS, set the CMake option `SUNDIALS_ENABLE_EXTERNAL_ADDONS` to ON
4. Build SUNDIALS as usual.

#### **SUNDIALS\_ENABLE\_EXTERNAL\_ADDONS**

Build SUNDIALS with any external addons that you have put in `SOLVER_DIR/external`.

Default: OFF

## 17.4 Testing the Build and Installation

If SUNDIALS was configured with any `EXAMPLES_ENABLE_<language>` options set to ON, then a set of regression tests can be run after building with the command:

```
make test
```

Additionally, if `SUNDIALS_ENABLE_EXAMPLES_INSTALL` is set to ON, then a set of smoke tests can be run after installing with the command:

```
make test_install
```

## 17.5 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 one of the `EXAMPLES_ENABLE_<language>` options to ON, and set `SUNDIALS_ENABLE_EXAMPLES_INSTALL` to ON. Along side the example sources and outputs, automatically generated `CMakeLists.txt` configuration files (and Makefile files if on Linux/Unix systems) are installed referencing the *installed* SUNDIALS headers and libraries.

Either the `CMakeLists.txt` file or the traditional Makefile may be used to build the examples and serve as a template for building user developed problems. 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.

## 17.6 Using SUNDIALS In Your Project

After installing SUNDIALS, building your application with SUNDIALS involves two steps: including the right header files and linking to the right libraries. Depending on what features of SUNDIALS that your application uses, the header files and libraries needed will vary. For example, if you want to use CVODE for serial computations you need the following includes:

```
#include <cvode/cvode.h>
#include <nvector/nvector_serial.h>
```

and must link to `libsundials_cvode` and `libsundials_nvecserial`. If you wanted to use CVODE with the GMRES linear solver and the CUDA NVector, you need the following includes:

```
#include <cvode/cvode.h>
#include <nvector/nvector_cuda.h>
#include <sunlinsol/sunlinsol_spgmr.h>
```

and must link to `libsundials_cvode`, `libsundials_nveccuda`, and `libsundials_sunlinsolspgmr`.

**Attention**

Added in version 7.0.0: All applications must also link to `libsundials_core`. For projects using SUNDIALS CMake targets (see section §17.6.1), this dependency is automatically included.

Refer to section §17.7 below or the documentations sections for the individual SUNDIALS packages and modules of interest for the proper includes and libraries to link against.

### 17.6.1 CMake Projects

For projects that use CMake, the SUNDIALS CMake package configuration file provides CMake targets for the consuming project. Use the CMake `find_package` command to search for the configuration file, `SUNDIALSConfig.cmake`, which is installed alongside a package version file, `SUNDIALSConfigVersion.cmake`, under the `INSTALL_DIR/SUNDIALS_INSTALL_CMAKEDIR` directory. The SUNDIALS CMake targets follow the same naming convention as the generated library binaries with the `libsundials_` prefix replaced by `SUNDIALS::`. For example, the exported target for `libsundials_cvode` is `SUNDIALS::cvode`. See section §17.7 for a complete list of CMake targets. The CMake code snippet below shows how a consuming project might leverage the SUNDIALS package configuration file to build against SUNDIALS in their own CMake project.

```

project(MyProject)

# Set the variable SUNDIALS_DIR to the SUNDIALS instdir.
# When using the cmake CLI command, this can be done like so:
#   cmake -D SUNDIALS_DIR=/path/to/sundials/installation

# Find any SUNDIALS version...
find_package(SUNDIALS REQUIRED)

# ... or find any version newer than some minimum...
find_package(SUNDIALS 7.1.0 REQUIRED)

# ... or find a version in a range
find_package(SUNDIALS 7.0.0...7.1.0 REQUIRED)

# To check if specific components are available in the SUNDIALS installation,
# use the COMPONENTS option followed by the desired target names
find_package(SUNDIALS REQUIRED COMPONENTS ccode nvecpetsc)

add_executable(myexec main.c)

# Link to SUNDIALS libraries through the exported targets.
# This is just an example, users should link to the targets appropriate
# for their use case.
target_link_libraries(myexec PUBLIC SUNDIALS::ccode SUNDIALS::nvecpetsc)

```

**Note**

Changed in version 7.1.0: A single version provided to `find_package` denotes the minimum version of SUNDIALS to look for, and any version equal or newer than what is specified will match. In prior versions `SUNDIALSConfig.cmake` required the version found to have the same major version number as the single version provided to `find_package`.

To accommodate installing both static and shared libraries simultaneously, targets are created with `_static` and `_shared` suffixes, respectively, and the un-suffixed target is an alias to the `_shared` version. For example, `SUNDIALS::ccode` is an alias to `SUNDIALS::ccode_shared` in this case. Projects that wish to use static libraries should use the `_static` version of the target when both library types are installed. When only static or shared libraries (not both) are installed the un-suffixed alias corresponds to the library type chosen at configuration time (see section §17.3.4).

## 17.7 Libraries and Header Files

As noted above, the SUNDIALS header files and libraries are installed under the `CMAKE_INSTALL_PREFIX` path in the `include` and `CMAKE_INSTALL_LIBDIR` subdirectories, respectively. The public header files are further organized into subdirectories under the `include` directory. The installed public header files and libraries are listed for reference in the sections below. Additionally, the exported CMake targets are also listed for projects using CMake (see section §17.6.1). The file extension `.LIB` used below is typically `.so`, `.dll`, or `.dylib` for shared libraries and `.a` or `.lib` for static libraries.

**Warning**

SUNDIALS installs some header files to `CMAKE_INSTALL_PREFIX/include/sundials/priv`. All of the header files in this directory are private and **should not be included in user code**. The private headers are subject to change without any notice and relying on them may break your code.

### 17.7.1 SUNDIALS Core

The core library contains the shared infrastructure utilized by SUNDIALS packages. All applications using SUNDIALS must link against the core library. For codes using the SUNDIALS CMake targets, the core target is automatically included as needed by other targets.

Table 17.1: The SUNDIALS core library, header, and CMake target

Libraries	<code>libsundials_core.LIB</code>
Headers	<code>sundials/sundials_core.h</code>
CMake target	<code>SUNDIALS::core</code>

The core header file is a convenient way to include all the header files that make up the SUNDIALS core infrastructure.

Table 17.2: Header files included by `sundials_core.h`

Headers	<code>sundials/sundials_adaptcontroller.h</code>
	<code>sundials/sundials_adjointstepper.h</code>
	<code>sundials/sundials_adjointcheckpointscheme.h</code>
	<code>sundials/sundials_config.h</code>
	<code>sundials/sundials_context.h</code>
	<code>sundials/sundials_domeigestimator.h</code>
	<code>sundials/sundials_errors.h</code>
	<code>sundials/sundials_iterative.h</code>
	<code>sundials/sundials_linearsolver.h</code>
	<code>sundials/sundials_logger.h</code>
	<code>sundials/sundials_math.h</code>
	<code>sundials/sundials_matrix.h</code>
	<code>sundials/sundials_memory.h</code>
	<code>sundials/sundials_nonlinearsolver.h</code>
	<code>sundials/sundials_nvector.h</code>
	<code>sundials/sundials_profiler.h</code>
	<code>sundials/sundials_types.h</code>
	<code>sundials/sundials_version.h</code>

For C++ applications, several convenience classes are provided for interacting with SUNDIALS objects. These can be accessed by including the C++ core header file.

Table 17.3: The SUNDIALS C++ core header file

Headers	<code>sundials/sundials_core.hpp</code>
---------	---

Like the C core header file, the C++ core header file is a convenient way to include all the header files for the core C++ classes.

**Warning**

Features in the `sundials::experimental` namespace are not yet part of the public API and are subject to change or removal without notice.

Table 17.4: Header files included by `sundials_core.hpp`

Headers	<code>sundials/sundials_context.hpp</code>
	<code>sundials/sundials_core.h</code>
	<code>sundials/sundials_linearsolver.hpp</code>
	<code>sundials/sundials_matrix.hpp</code>
	<code>sundials/sundials_memory.hpp</code>
	<code>sundials/sundials_nonlinearsolver.hpp</code>
	<code>sundials/sundials_nvector.hpp</code>
	<code>sundials/sundials_profiler.hpp</code>

When MPI support is enabled (`SUNDIALS_ENABLE_MPI` is ON), the following header file provides aliases between MPI data types and SUNDIALS types. The alias `MPI_SUNREALTYPE` is one of `MPI_FLOAT`, `MPI_DOUBLE`, or `MPI_LONG_DOUBLE` depending on the value of `SUNDIALS_PRECISION`. The alias `MPI_SUNINDEXTYPE` is either `MPI_INT32_T` or `MPI_INT64_T` depending on the value of `SUNDIALS_INDEX_SIZE`.

Table 17.5: Header file defining aliases between SUNDIALS and MPI data types

Headers	<code>sundials/sundials_mpi_types.h</code>
---------	--

When XBraid support is enabled (`SUNDIALS_ENABLE_XBRAID` is ON), the following header file defines types and functions for interfacing SUNDIALS with XBraid.

Table 17.6: SUNDIALS header for interfacing with XBraid

Headers	<code>sundials/sundials_xbraid.h</code>
---------	---

## 17.7.2 SUNDIALS Packages

### 17.7.2.1 CVODE

To use the `CVODE` package, include the header file and link to the library given below.

Table 17.7: CVODE library, header file, and CMake target

Libraries	<code>libsundials_cvode.LIB</code>
Headers	<code>cvode/cvode.h</code>
CMake target	<code>SUNDIALS::cvode</code>

The CVODE header file includes the files below which define functions, types, and constants for the CVODE linear solver interface and using projection methods with CVODE.

Table 17.8: Additional header files included by `cvode.h`

Headers	<code>cvode/cvode_ls.h</code> <code>cvode/cvode_proj.h</code>
---------	--

CVODE provides a specialized linear solver module for diagonal linear systems. Include the header file below to access the related functions.

Table 17.9: CVODE diagonal linear solver

Headers	<code>cvode/cvode_diag.h</code>
---------	---------------------------------

For problems in which the user cannot define a more effective, problem-specific preconditioner for Krylov iterative linear solvers, CVODE provides banded (`bandpre`) and band-block-diagonal (`bbdpre`) preconditioner modules. Include the header files below to access the related functions.

Table 17.10: CVODE preconditioner modules

Headers	<code>cvode/cvode_bandpre.h</code> <code>cvode/cvode_bbdpre.h</code>
---------	---

### 17.7.2.2 CVODES

To use the [CVODES](#) package, include the header file and link to the library given below.

#### Warning

CVODES is a superset of CVODE and defines the same functions as provided by CVODE. As such, applications should not link to both CVODES and CVODE.

Table 17.11: CVODES library, header file, and CMake target

Libraries	<code>libsundials_cvodes.LIB</code>
Headers	<code>cvodes/cvodes.h</code>
CMake target	<code>SUNDIALS::cvodes</code>

The CVODES header file includes the files below which define functions, types, and constants for the CVODES linear solver interface and using projection methods with CVODES.

Table 17.12: Additional header files included by `cvodes.h`

Headers	<code>cvodes/cvodes_ls.h</code> <code>cvodes/cvodes_proj.h</code>
---------	--

CVODES provides a specialized linear solver module for diagonal linear systems. Include the header file below to access the related functions.

Table 17.13: CVODES diagonal linear solver

Headers	<code>cvodes/cvodes_diag.h</code>
---------	-----------------------------------



For problems in which the user cannot define a more effective, problem-specific preconditioner for Krylov iterative linear solvers, CVODES provides banded (`bandpre`) and band-block-diagonal (`bbdpre`) preconditioner modules. Include the header files below to access the related functions.

Table 17.14: CVODES preconditioner modules

Headers	<code>cvodes/cvodes_bandpre.h</code> <code>cvodes/cvodes_bbdpre.h</code>
---------	---

### 17.7.2.3 ARKODE

To use the [ARKODE](#) package, link to the library below and include the header file for the desired module.

Table 17.15: ARKODE library, header files, and CMake target

Libraries	<code>libsundials_arkode.LIB</code>
Headers	<code>arkode/arkode_arkstep.h</code> <code>arkode/arkode_erkstep.h</code> <code>arkode/arkode_forcingstep.h</code> <code>arkode/arkode_lsrkstep.h</code> <code>arkode/arkode_mristep.h</code> <code>arkode/arkode_splittingstep.h</code> <code>arkode/arkode_sprkstep.h</code>
CMake target	<code>SUNDIALS::arkode</code>

The ARKODE module header files include the header file for the shared ARKODE interface functions, constants, and types (`arkode.h`). As appropriate, the module header files also include the ARKODE linear solver interface as well as the header files defining method coefficients.

Table 17.16: Additional header files included by `arkode_*step.h` header files

Headers	<code>arkode/arkode.h</code> <code>arkode/arkode_butcher.h</code> <code>arkode/arkode_butcher_dirk.h</code> <code>arkode/arkode_butcher_erk.h</code> <code>arkode/arkode_ls.h</code> <code>arkode/arkode_sprk.h</code>
---------	---

For problems in which the user cannot define a more effective, problem-specific preconditioner for Krylov iterative linear solvers, ARKODE provides banded (`bandpre`) and band-block-diagonal (`bbdpre`) preconditioner modules. Include the header files below to access the related functions.

Table 17.17: ARKODE preconditioner modules

Headers	<code>arkode/arkode_bandpre.h</code> <code>arkode/arkode_bbdpre.h</code>
---------	---

When XBraid support is enabled (`SUNDIALS_ENABLE_XBRAID` is ON), include the ARKODE-XBraid interface header file and link to the interface library given below to use ARKODE and XBraid together.

Table 17.18: ARKODE library, header, and CMake target for interfacing with XBraid

Libraries	libsundials_arkode_xbraid.LIB
Headers	arkode/arkode_xbraid.h
CMake target	SUNDIALS::arkode_xbraid

#### 17.7.2.4 IDA

To use the [IDA](#) package, include the header file and link to the library given below.

Table 17.19: IDA library, header file, and CMake target

Libraries	libsundials_ida.LIB
Headers	ida/ida.h
CMake target	SUNDIALS::ida

The IDA header file includes the header file below which defines functions, types, and constants for the IDA linear solver interface.

Table 17.20: Additional header files included by `ida.h`

Headers	ida/ida_ls.h
---------	--------------

For problems in which the user cannot define a more effective, problem-specific preconditioner for Krylov iterative linear solvers, IDA provides a band-block-diagonal (`bbdpre`) preconditioner module. Include the header file below to access the related functions.

Table 17.21: IDA preconditioner modules

Headers	ida/ida_bbdpre.h
---------	------------------

#### 17.7.2.5 IDAS

To use the [IDAS](#) package, include the header file and link to the library given below.

##### Warning

IDAS is a superset of IDA and defines the same functions as provided by IDA. As such, applications should not link to both IDAS and IDA.

Table 17.22: IDAS library, header file, and CMake target

Libraries	libsundials_idas.LIB
Headers	idas/idas.h
CMake target	SUNDIALS::idas

The IDAS header file includes the header file below which defines functions, types, and constants for the IDAS linear solver interface.

Table 17.23: Additional header files included by `idas.h`

Headers	<code>idas/idas_ls.h</code>
---------	-----------------------------

For problems in which the user cannot define a more effective, problem-specific preconditioner for Krylov iterative linear solvers, IDAS provides a band-block-diagonal (`bddpre`) preconditioner module. Include the header file below to access the related functions.

Table 17.24: IDAS preconditioner modules

Headers	<code>idas/idas_bddpre.h</code>
---------	---------------------------------

### 17.7.2.6 KINSOL

To use the [KINSOL](#) package, include the header file and link to the library given below.

Table 17.25: KINSOL library, header file, and CMake target

Libraries	<code>libsundials_kinsol.LIB</code>
Headers	<code>kinsol/kinsol.h</code>
CMake target	<code>SUNDIALS::kinsol</code>

The KINSOL header file includes the header file below which defines functions, types, and constants for the KINSOL linear solver interface.

Table 17.26: Additional header files included by `kinsol.h`

Headers	<code>kinsol/kinsol_ls.h</code>
---------	---------------------------------

For problems in which the user cannot define a more effective, problem-specific preconditioner for Krylov iterative linear solvers, KINSOL provides a band-block-diagonal (`bddpre`) preconditioner module. Include the header file below to access the related functions.

Table 17.27: KINSOL preconditioner modules

Headers	<code>kinsol/kinsol_bddpre.h</code>
---------	-------------------------------------

## 17.7.3 Vectors

### 17.7.3.1 Serial

To use the [serial NVector](#), include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the serial NVector is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.28: The serial NVector library, header file, and CMake target

Libraries	<code>libsundials_nvecserial.LIB</code>
Headers	<code>nvector/nvector_serial.h</code>
CMake target	<code>SUNDIALS::nvecserial</code>

### 17.7.3.2 ManyVector

To use the *ManyVector NVector*, include the header file and link to the library given below.

Table 17.29: The ManyVector NVector library, header file, and CMake target

Libraries	libsundials_nvecmanyvector.LIB
Headers	nvector/nvector_manyvector.h
CMake target	SUNDIALS::nvecmanyvector

### 17.7.3.3 Parallel (MPI)

To use the *parallel (MPI) NVector*, include the header file and link to the library given below.

Table 17.30: The parallel (MPI) NVector library, header file, and CMake target

Libraries	libsundials_nvecparallel.LIB
Headers	nvector/nvector_parallel.h
CMake target	SUNDIALS::nvecparallel

### 17.7.3.4 MPI ManyVector

To use the *MPI ManyVector NVector*, include the header file and link to the library given below.

Table 17.31: The MPI ManyVector NVector library, header file, and CMake target

Libraries	libsundials_nvecmpimanyvector.LIB
Headers	nvector/nvector_mpimanyvector.h
CMake target	SUNDIALS::nvecmpimanyvector

### 17.7.3.5 MPI+X

To use the *MPI+X NVector*, include the header file and link to the library given below.

Table 17.32: The MPI+X NVector library, header file, and CMake target

Libraries	libsundials_nvecmpiplusx.LIB
Headers	nvector/nvector_mpiplusx.h
CMake target	SUNDIALS::nvecmpiplusx

### 17.7.3.6 OpenMP

To use the *OpenMP NVector*, include the header file and link to the library given below.

Table 17.33: The OpenMP NVector library, header file, and CMake target

Libraries	libsundials_nvecopenmp.LIB
Headers	nvector/nvector_openmp.h
CMake target	SUNDIALS::nvecopenmp

### 17.7.3.7 OpenMPDEV

To use the *OpenMP device offload NVector*, include the header file and link to the library given below.

Table 17.34: The OpenMP device offload NVector library, header file, and CMake target

Libraries	libsundials_nvecopenmpdev.LIB
Headers	nvector/nvector_openmpdev.h
CMake target	SUNDIALS::nvecopenmpdev

### 17.7.3.8 PThreads

To use the *POSIX Threads NVector*, include the header file and link to the library given below.

Table 17.35: The POSIX Threads NVector library, header file, and CMake target

Libraries	libsundials_nvecpthreads.LIB
Headers	nvector/nvector_pthreads.h
CMake target	SUNDIALS::nvecpthreads

### 17.7.3.9 hypre (ParHyp)

To use the *hypre (ParHyp) NVector*, include the header file and link to the library given below.

Table 17.36: The hypre (ParHyp) NVector library, header file, and CMake target

Libraries	libsundials_nvecparhyp.LIB
Headers	nvector/nvector_parhyp.h
CMake target	SUNDIALS::nvecparhyp

### 17.7.3.10 PETSc

To use the *PETSc NVector*, include the header file and link to the library given below.

Table 17.37: The PETSc NVector library, header file, and CMake target

Libraries	libsundials_nvecpetsc.LIB
Headers	nvector/nvector_petsc.h
CMake target	SUNDIALS::nvecpetsc

### 17.7.3.11 CUDA

To use the *CUDA NVector*, include the header file and link to the library given below.

Table 17.38: The CUDA NVector library, header file, and CMake target

Libraries	libsundials_nveccuda.LIB
Headers	nvector/nvector_cuda.h
CMake target	SUNDIALS::nveccuda

### 17.7.3.12 HIP

To use the *HIP NVector*, include the header file and link to the library given below.

Table 17.39: The HIP NVector library, header file, and CMake target

Libraries	libsundials_nvechip.LIB
Headers	nvector/nvector_hip.h
CMake target	SUNDIALS::nvechip

### 17.7.3.13 RAJA

To use the *RAJA NVector*, include the header file and link to the library given below for the desired backend.

Table 17.40: The RAJA NVector libraries, header file, and CMake targets

Libraries	libsundials_nveccudaraja.LIB
	libsundials_nvechipraja.LIB
	libsundials_nvecsyclraja.LIB
Headers	nvector/nvector_raja.h
CMake target	SUNDIALS::nveccudaraja
	SUNDIALS::nvechipraja
	SUNDIALS::nvecsyclraja

### 17.7.3.14 SYCL

To use the *SYCL NVector*, include the header file and link to the library given below.

Table 17.41: The SYCL NVector library, header file, and CMake target

Libraries	libsundials_nvecsycl.LIB
Headers	nvector/nvector_sycl.h
CMake target	SUNDIALS::nvecsycl

### 17.7.3.15 Trilinos (Tpetra)

To use the *Trilinos (Tpetra) NVector*, include the header file and link to the library given below.

Table 17.42: The Trilinos (Tpetra) NVector library, header file, and CMake target

Libraries	libsundials_nvectrilinos.LIB
Headers	nvector/nvector_trilinos.h
CMake target	SUNDIALS::nvectrilinos

### 17.7.3.16 Kokkos

To use the *Kokkos NVector*, include the header file and link to the library given below.

Table 17.43: The Kokkos NVector library, header file, and CMake target

Headers	nvector/nvector_kokkos.hpp
CMake target	SUNDIALS::nveckokkos

## 17.7.4 Matrices

### 17.7.4.1 Banded

To use the *banded SUNMatrix*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the banded SUNMatrix is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.44: The banded SUNMatrix library, header file, and CMake target

Libraries	libsundials_sunmatrixband.LIB
Headers	sunmatrix/sunmatrix_band.h
CMake target	SUNDIALS::sunmatrixband

### 17.7.4.2 cuSPARSE

To use the *cuSPARSE SUNMatrix*, include the header file and link to the library given below.

Table 17.45: The cuSPARSE SUNMatrix library, header file, and CMake target

Libraries	libsundials_sunmatrixcusparse.LIB
Headers	sunmatrix/sunmatrix_cusparse.h
CMake target	SUNDIALS::sunmatrixcusparse

### 17.7.4.3 Dense

To use the *dense SUNMatrix*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the dense SUNMatrix is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.46: The dense SUNMatrix library, header file, and CMake target

Libraries	libsundials_sunmatrixdense.LIB
Headers	sunmatrix/sunmatrix_dense.h
CMake target	SUNDIALS::sunmatrixdense

#### 17.7.4.4 Ginkgo

To use the *Ginkgo SUNMatrix* or *Ginkgo Batch SUNMatrix*, include the corresponding header file given below.

Table 17.47: The Ginkgo and Ginkgo Batch SUNMatrix header files and CMake target

Headers	sunmatrix/sunmatrix_ginkgo.hpp sunmatrix/sunmatrix_ginkgobatch.hpp
CMake target	SUNDIALS::sunmatrixginkgo

#### 17.7.4.5 KokkosKernels Dense

To use the *KokkosKernels dense SUNMatrix*, include the header file given below.

Table 17.48: The dense KokkosKernels SUNMatrix library, header file, and CMake target

Headers	sunmatrix/sunmatrix_kokkosdense.hpp
CMake target	SUNDIALS::sunmatrixkokkosdense

#### 17.7.4.6 MAGMA Dense

To use the *MAGMA dense SUNMatrix*, include the header file and link to the library given below.

Table 17.49: The dense MAGMA SUNMatrix library, header file, and CMake target

Libraries	libsundials_sunmatrixmagmadense.LIB
Headers	sunmatrix/sunmatrix_magmadense.h
CMake target	SUNDIALS::sunmatrixmagmadense

#### 17.7.4.7 oneMKL Dense

To use the *oneMKL dense SUNMatrix*, include the header file and link to the library given below.

Table 17.50: The dense oneMKL SUNMatrix library, header file, and CMake target

Libraries	libsundials_sunmatrixonemkldense.LIB
Headers	sunmatrix/sunmatrix_onemkldense.h
CMake target	SUNDIALS::sunmatrixonemkldense



### 17.7.4.8 Sparse

To use the *sparse SUNMatrix*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the sparse SUNMatrix is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.51: The sparse SUNMatrix library, header file, and CMake target

Libraries	libsundials_sunmatrixsparse.LIB
Headers	sunmatrix/sunmatrix_sparse.h
CMake target	SUNDIALS::sunmatrixsparse

### 17.7.4.9 SuperLU\_DIST (SLUNRloc)

To use the *SuperLU\_DIST (SLUNRloc) SUNMatrix*, include the header file and link to the library given below.

Table 17.52: The SuperLU\_DIST (SLUNRloc) SUNMatrix library, header file, and CMake target

Libraries	libsundials_sunmatrixslunrloc.LIB
Headers	sunmatrix/sunmatrix_slunrloc.h
CMake target	SUNDIALS::sunmatrixslunrloc

## 17.7.5 Linear Solvers

### 17.7.5.1 Banded

To use the *banded SUNLinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the banded SUNLinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.53: The banded SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolband.LIB
Headers	sunlinsol/sunlinsol_band.h
CMake target	SUNDIALS::sunlinsolband

### 17.7.5.2 cuSPARSE Batched QR

To use the *cuSPARSE batched QR SUNLinearSolver*, include the header file and link to the library given below.

Table 17.54: The cuSPARSE batched QR SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolcusolversp.LIB
Headers	sunlinsol/sunlinsol_cusolversp_batchqr.h
CMake target	SUNDIALS::sunlinsolcusolversp

### 17.7.5.3 Dense

To use the *dense SUNLinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the dense SUNLinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.55: The dense SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsoldense.LIB
Headers	sunlinsol/sunlinsol_dense.h
CMake target	SUNDIALS::sunlinsoldense

### 17.7.5.4 Ginkgo

To use the *Ginkgo SUNLinearSolver* or *Ginkgo Batch SUNLinearSolver*, include the corresponding header file given below.

Table 17.56: The Ginkgo and Ginkgo Batch SUNLinearSolver header files and CMake target

Headers	sunlinsol/sunlinsol_ginkgo.hpp
	sunlinsol/sunlinsol_ginkgobatch.hpp
CMake target	SUNDIALS::sunlinsolginkgo

### 17.7.5.5 KLU

To use the *KLU SUNLinearSolver*, include the header file and link to the library given below.

Table 17.57: The KLU SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolklu.LIB
Headers	sunlinsol/sunlinsol_klu.h
CMake target	SUNDIALS::sunlinsolklu

### 17.7.5.6 KokkosKernels Dense

To use the *KokkosKernels dense SUNLinearSolver*, include the header file given below.

Table 17.58: The KokkosKernels dense SUNLinearSolver header file and CMake target

Headers	sunlinsol/sunlinsol_kokkosdense.hpp
CMake target	SUNDIALS::sunlinsolkokkosdense

### 17.7.5.7 LAPACK Banded

To use the *LAPACK banded SUNLinearSolver*, include the header file and link to the library given below.

Table 17.59: The LAPACK banded SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsollapackband.LIB
Headers	sunlinsol/sunlinsol_lapackband.h
CMake target	SUNDIALS::sunlinsollapackband

### 17.7.5.8 LAPACK Dense

To use the *LAPACK dense SUNLinearSolver*, include the header file and link to the library given below.

Table 17.60: The LAPACK dense SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsollapackdense.LIB
Headers	sunlinsol/sunlinsol_lapackdense.h
CMake target	SUNDIALS::sunlinsollapackdense

### 17.7.5.9 MAGMA Dense

To use the *MAGMA dense SUNLinearSolver*, include the header file and link to the library given below.

Table 17.61: The MAGMA dense SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolmagmadense.LIB
Headers	sunlinsol/sunlinsol_magmadense.h
CMake target	SUNDIALS::sunlinsolmagmadense

### 17.7.5.10 oneMKL Dense

To use the *oneMKL dense SUNLinearSolver*, include the header file and link to the library given below.

Table 17.62: The oneMKL dense SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolonemkldense.LIB
Headers	sunlinsol/sunlinsol_onemkldense.h
CMake target	SUNDIALS::sunlinsolonemkldense

### 17.7.5.11 Preconditioned Conjugate Gradient (PCG)

To use the *PCG SUNLinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the PCG SUNLinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.63: The PCG SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolpcg.LIB
Headers	sunlinsol/sunlinsol_pcg.h
CMake target	SUNDIALS::sunlinsolpcg

#### 17.7.5.12 Scaled, Preconditioned Bi-Conjugate Gradient, Stabilized (SPBCGS)

To use the *SPBCGS SUNLinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the SPBCGS SUNLinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.64: The SPBCGS SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolspbcgs.LIB
Headers	sunlinsol/sunlinsol_spbcgs.h
CMake target	SUNDIALS::sunlinsolspbcgs

#### 17.7.5.13 Scaled, Preconditioned, Flexible, Generalized Minimum Residual (SPFGMR)

To use the *SPFGMR SUNLinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the SPFGMR SUNLinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.65: The SPFGMR SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolspfgmr.LIB
Headers	sunlinsol/sunlinsol_spfgmr.h
CMake target	SUNDIALS::sunlinsolspfgmr

#### 17.7.5.14 Scaled, Preconditioned, Generalized Minimum Residual (SPGMR)

To use the *SPGMR SUNLinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the SPGMR SUNLinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.66: The SPGMR SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolspgmr.LIB
Headers	sunlinsol/sunlinsol_spgmr.h
CMake target	SUNDIALS::sunlinsolspgmr

### 17.7.5.15 Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual (SPTFQMR)

To use the *SPTFQMR SUNLinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages or the KINSOL package, the SPTFQMR SUNLinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.67: The SPTFQMR SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolsptfqr.LIB
Headers	sunlinsol/sunlinsol_sptfqr.h
CMake target	SUNDIALS::sunlinsolsptfqr

### 17.7.5.16 SuperLU\_DIST

To use the *SuperLU\_DIST SUNLinearSolver*, include the header file and link to the library given below.

Table 17.68: The SuperLU\_DIST SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolsuperludist.LIB
Headers	sunlinsol/sunlinsol_superludist.h
CMake target	SUNDIALS::sunlinsolsuperludist

### 17.7.5.17 SuperLU\_MT

To use the *SuperLU\_MT SUNLinearSolver*, include the header file and link to the library given below.

Table 17.69: The SuperLU\_MT SUNLinearSolver library, header file, and CMake target

Libraries	libsundials_sunlinsolsuperlumt.LIB
Headers	sunlinsol/sunlinsol_superlumt.h
CMake target	SUNDIALS::sunlinsolsuperlumt

## 17.7.6 Nonlinear Solvers

### 17.7.6.1 Newton

To use the *Newton SUNNonlinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages, the Newton SUNNonlinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.70: The Newton SUNNonlinearSolver library, header file, and CMake target

Libraries	libsundials_sunnonlinsolnewton.LIB
Headers	sunnonlinsol/sunnonlinsol_newton.h
CMake target	SUNDIALS::sunnonlinsolnewton

### 17.7.6.2 Fixed-point

To use the *fixed-point SUNNonlinearSolver*, include the header file and link to the library given below.

When using SUNDIALS time integration packages, the fixed-point SUNNonlinearSolver is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.71: The Fixed-point SUNNonlinearSolver library, header file, and CMake target

Libraries	libsundials_sunnonlinsolfixedpoint.LIB
Headers	sunnonlinsol/sunnonlinsol_fixedpoint.h
CMake target	SUNDIALS::sunnonlinsolfixedpoint

### 17.7.6.3 PETSc SNES

To use the *PETSc SNES SUNNonlinearSolver*, include the header file and link to the library given below.

Table 17.72: The PETSc SNES SUNNonlinearSolver library, header file, and CMake target

Libraries	libsundials_sunnonlinsolpetscsnes.LIB
Headers	sunnonlinsol/sunnonlinsol_petscsnes.h
CMake target	SUNDIALS::sunnonlinsolpetscsnes

## 17.7.7 Memory Helpers

### 17.7.7.1 System

When using SUNDIALS time integration packages or the KINSOL package, the system SUNMemoryHelper is bundled with the package library and it is not necessary to link to the library below when using those packages.

Table 17.73: SUNDIALS system memory helper header file

Headers	sunmemory/sunmemory_system.h
---------	------------------------------

### 17.7.7.2 CUDA

To use the *CUDA SUNMemoryHelper*, include the header file given below when using a CUDA-enabled NVector or SUNMatrix.

Table 17.74: SUNDIALS CUDA memory helper header file

Headers	sunmemory/sunmemory_cuda.h
---------	----------------------------

### 17.7.7.3 HIP

To use the *HIP SUNMemoryHelper*, include the header file given below when using a HIP-enabled NVector or SUNMatrix.

Table 17.75: SUNDIALS HIP memory helper header file

Headers	<code>sunmemory/sunmemory_hip.h</code>
---------	--

#### 17.7.7.4 SYCL

To use the *SYCL SUNMemoryHelper*, include the header file given below when using a SYCL-enabled NVector or SUNMatrix.

Table 17.76: SUNDIALS SYCL memory helper header file

Headers	<code>sunmemory/sunmemory_sycl.h</code>
---------	---

### 17.7.8 Execution Policies

#### 17.7.8.1 CUDA

When using a CUDA-enabled NVector or SUNMatrix, include the header file below to access the CUDA execution policy C++ classes.

Table 17.77: SUNDIALS CUDA execution policies header file

Headers	<code>sundials/sundials_cuda_policies.hpp</code>
---------	--

#### 17.7.8.2 HIP

When using a HIP-enabled NVector or SUNMatrix, include the header file below to access the HIP execution policy C++ classes.

Table 17.78: SUNDIALS HIP execution policies header file

Headers	<code>sundials/sundials_hip_policies.hpp</code>
---------	---

#### 17.7.8.3 SYCL

When using a SYCL-enabled NVector or SUNMatrix, include the header file below to access the SYCL execution policy C++ classes.

Table 17.79: SUNDIALS SYCL execution policies header file

Headers	<code>sundials/sundials_sycl_policies.hpp</code>
---------	--

### 17.7.9 Adjoint Sensitivity Checkpointing

#### 17.7.9.1 Fixed ASA Checkpointing

For fixed-interval adjoint checkpointing, include the header file below:

Table 17.80: SUNDIALS fixed adjoint checkpointing header files

Headers	sunadjointcheckpointscheme/sunadjointcheckpointscheme_fixed.h
---------	---

## 17.7.10 Dominant Eigenvalue Estimation

### 17.7.10.1 Power Iteration

To use the *Power iteration SUNDomEigEstimator*, include the header file and link to the library given below.

Table 17.81: The SUNDIALS Power iteration SUNDomEigEstimator library, header file, and CMake target

Libraries	libsundials_sundomeigestpower.LIB
Headers	sundomeigest/sundomeigest_power.h
CMake target	SUNDIALS::sundomeigestpower

### 17.7.10.2 Arnoldi Iteration

To use the *Arnoldi iteration SUNDomEigEstimator*, include the header file and link to the library given below.

Table 17.82: The SUNDIALS Arnoldi iteration SUNDomEigEstimator library, header file, and CMake target

Libraries	libsundials_sundomeigestarnoldi.LIB
Headers	sundomeigest/sundomeigest_arnoldi.h
CMake target	SUNDIALS::sundomeigestarnoldi



# Chapter 18

## ARKODE Constants

Below we list all input and output constants used by the main solver, timestepper, and linear solver modules, together with a short description of their meaning. [Table 18.1](#) contains the ARKODE input constants, and [Table 18.2](#) contains the ARKODE output constants.

Table 18.1: ARKODE input constants

<b>Shared input constants</b>	
ARK_NORMAL	Solver should return at a specified output time.
ARK_ONE_STEP	Solver should return after each successful step.
<b>Full right-hand side evaluation constants</b>	
ARK_FULLRHS_START	Calling the full right-hand side function at the start of the integration.
ARK_FULLRHS_END	Calling the full right-hand side function at the end of a step.
ARK_FULLRHS_OTHER	Calling the full right-hand side function at the some other point e.g., for dense output.
<b>Interpolation module input constants</b>	
ARK_INTERP_NONE	Disables polynomial interpolation for dense output.
ARK_INTERP_HERMITE	Specifies use of the Hermite polynomial interpolation module (for non-stiff problems).
ARK_INTERP_LAGRANGE	Specifies use of the Lagrange polynomial interpolation module (for stiff problems).
ARK_INTERP_MAX_DEGREE	Maximum possible interpolating polynomial degree.
<b>Relaxation module input constants</b>	
ARK_RELAX_BRENT	Specifies Brent's method as the relaxation nonlinear solver.
ARK_RELAX_NEWTON	Specifies Newton's method as the relaxation nonlinear solver.
<b>Default explicit Butcher tables</b>	
ARKSTEP_DEFAULT_ERK_1	Use ARKStep's default first-order ERK method <a href="#">ARKODE_FORWARD_EULER_1_1</a> .
ARKSTEP_DEFAULT_ERK_2	Use ARKStep's default second-order ERK method <a href="#">ARKODE_RALSTON_3_1_2</a> .
ARKSTEP_DEFAULT_ERK_3	Use ARKStep's default third-order ERK method <a href="#">ARKODE_BOGACKI_SHAMPINE_4_2_3</a> .

continues on next page

Table 18.1 – continued from previous page

ARKSTEP_DEFAULT_ERK_4	Use ARKStep's default fourth-order ERK method <a href="#">ARKODE_SOFRONIOU_SPALETTA_5_3_4</a> .
ARKSTEP_DEFAULT_ERK_5	Use ARKStep's default fifth-order ERK method <a href="#">ARKODE_TSI-TOURAS_7_4_5</a> .
ARKSTEP_DEFAULT_ERK_6	Use ARKStep's default sixth-order ERK method <a href="#">ARKODE_VERNER_9_5_6</a> .
ARKSTEP_DEFAULT_ERK_7	Use ARKStep's default seventh-order ERK method <a href="#">ARKODE_VERNER_10_6_7</a> .
ARKSTEP_DEFAULT_ERK_8	Use ARKStep's default eighth-order ERK method <a href="#">ARKODE_VERNER_13_7_8</a> .
ARKSTEP_DEFAULT_ERK_9	Use ARKStep's default ninth-order ERK method <a href="#">ARKODE_VERNER_16_8_9</a> .
ERKSTEP_DEFAULT_1	Use ERKStep's default first-order ERK method <a href="#">ARKODE_FORWARD_EULER_1_1</a> .
ERKSTEP_DEFAULT_2	Use ERKStep's default second-order ERK method <a href="#">ARKODE_RAL-STON_3_1_2</a> .
ERKSTEP_DEFAULT_3	Use ERKStep's default third-order ERK method <a href="#">ARKODE_BO-GACKI_SHAMPINE_4_2_3</a> .
ERKSTEP_DEFAULT_4	Use ERKStep's default fourth-order ERK method <a href="#">ARKODE_SOFRONIOU_SPALETTA_5_3_4</a> .
ERKSTEP_DEFAULT_5	Use ERKStep's default fifth-order ERK method <a href="#">ARKODE_TSI-TOURAS_7_4_5</a> .
ERKSTEP_DEFAULT_6	Use ERKStep's default sixth-order ERK method <a href="#">ARKODE_VERNER_9_5_6</a> .
ERKSTEP_DEFAULT_7	Use ERKStep's default seventh-order ERK method <a href="#">ARKODE_VERNER_10_6_7</a> .
ERKSTEP_DEFAULT_8	Use ERKStep's default eighth-order ERK method <a href="#">ARKODE_VERNER_13_7_8</a> .
ERKSTEP_DEFAULT_9	Use ERKStep's default ninth-order ERK method <a href="#">ARKODE_VERNER_16_8_9</a> .
<b>Default implicit Butcher tables</b>	
ARKSTEP_DEFAULT_DIRK_1	Use ARKStep's default first-order DIRK method <a href="#">ARKODE_BACKWARD_EULER_1_1</a> .
ARKSTEP_DEFAULT_DIRK_2	Use ARKStep's default second-order DIRK method <a href="#">ARKODE_ARK2_DIRK_3_1_2</a> .
ARKSTEP_DEFAULT_DIRK_3	Use ARKStep's default third-order DIRK method <a href="#">ARKODE_ES-DIRK325L2SA_5_2_3</a> .
ARKSTEP_DEFAULT_DIRK_4	Use ARKStep's default fourth-order DIRK method <a href="#">ARKODE_ES-DIRK436L2SA_6_3_4</a> .
ARKSTEP_DEFAULT_DIRK_5	Use ARKStep's default fifth-order DIRK method <a href="#">ARKODE_ES-DIRK547L2SA2_7_4_5</a> .
<b>Default ImEx Butcher tables</b>	
ARKSTEP_DEFAULT_ARK_ETABLE_2 & ARKSTEP_DEFAULT_ARK_ITABLE_2	Use ARKStep's default second-order ARK method (ARKODE_ARK2_ERK_3_1_2 and ARKODE_ARK2_DIRK_3_1_2).
ARKSTEP_DEFAULT_ARK_ETABLE_3 & ARKSTEP_DEFAULT_ARK_ITABLE_3	Use ARKStep's default third-order ARK method (ARKODE_ARK324L2SA_ERK_4_2_3 and ARKODE_ARK324L2SA-DIRK_4_2_3).
ARKSTEP_DEFAULT_ARK_ETABLE_4 & ARKSTEP_DEFAULT_ARK_ITABLE_4	Use ARKStep's default fourth-order ARK method (ARKODE_ARK436L2SA_ERK_6_3_4 and ARKODE_ARK436L2SA-DIRK_6_3_4).

continues on next page

Table 18.1 – continued from previous page

ARKSTEP_DEFAULT_ARK_ETABLE_5 & ARKSTEP_DEFAULT_ARK_ITABLE_5	Use ARKStep’s default fifth-order ARK method (ARKODE_ARK548L2SA_ERK_8_4_5 and ARKODE_ARK548L2SA_DIRK_8_4_5).
<b>LSRK method types</b>	
ARKODE_LSRK_RKC_2	2nd order Runge-Kutta-Chebyshev (RKC) method <a href="#">ARKODE_LSRK_RKC_2</a>
ARKODE_LSRK_RKL_2	2nd order Runge-Kutta-Legendre (RKL) method <a href="#">ARKODE_LSRK_RKL_2</a>
ARKODE_LSRK_SSP_S_2	Optimal 2nd order s-stage SSP RK method <a href="#">ARKODE_LSRK_SSP_S_2</a>
ARKODE_LSRK_SSP_S_3	Optimal 3rd order s-stage SSP RK method <a href="#">ARKODE_LSRK_SSP_S_3</a>
ARKODE_LSRK_SSP_10_4	Optimal 4th order 10-stage SSP RK method <a href="#">ARKODE_LSRK_SSP_10_4</a>
<b>MRI method types</b>	
MRISTEP_EXPLICIT	Use an explicit (at the slow time scale) MRI method.
MRISTEP_IMPLICIT	Use an implicit (at the slow time scale) MRI method.
MRISTEP_IMEX	Use an ImEx (at the slow time scale) MRI method.
<b>Default MRI coupling tables</b>	
MRISTEP_DEFAULT_EXPL_1	Use MRISTep’s default 1st-order explicit method (ARKODE_MRI_GARK_FORWARD_EULER).
MRISTEP_DEFAULT_EXPL_2	Use MRISTep’s default 2nd-order explicit method (ARKODE_MRI_GARK_ERK22b).
MRISTEP_DEFAULT_EXPL_3	Use MRISTep’s default 3rd-order explicit method (ARKODE_MIS_KW3).
MRISTEP_DEFAULT_EXPL_4	Use MRISTep’s default 4th-order explicit method (ARKODE_MRI_GARK_ERK45a).
MRISTEP_DEFAULT_EXPL_2_AD	Use MRISTep’s default 2nd-order adaptive explicit method (ARKODE_MRI_GARK_ERK22a).
MRISTEP_DEFAULT_EXPL_3_AD	Use MRISTep’s default 3rd-order adaptive explicit method (ARKODE_MRI_GARK_ERK33a).
MRISTEP_DEFAULT_EXPL_4_AD	Use MRISTep’s default 4th-order adaptive explicit method (ARKODE_MRI_GARK_ERK45a).
MRISTEP_DEFAULT_EXPL_5_AD	Use MRISTep’s default 5th-order adaptive explicit method (ARKODE_MERK54).
MRISTEP_DEFAULT_IMPL_SD_1	Use MRISTep’s default 1st-order solve-decoupled implicit method (ARKODE_MRI_GARK_BACKWARD_EULER).
MRISTEP_DEFAULT_IMPL_SD_2	Use MRISTep’s default 2nd-order solve-decoupled implicit method (ARKODE_MRI_GARK_IRK21a).
MRISTEP_DEFAULT_IMPL_SD_3	Use MRISTep’s default 3rd-order solve-decoupled implicit method (ARKODE_MRI_GARK_ESDIRK34a).
MRISTEP_DEFAULT_IMPL_SD_4	Use MRISTep’s default 4th-order solve-decoupled implicit method (ARKODE_MRI_GARK_ESDIRK46a).
MRISTEP_DEFAULT_IMEX_SD_1	Use MRISTep’s default 1st-order solve-decoupled ImEx method (ARKODE_IMEX_MRI_GARK_EULER).
MRISTEP_DEFAULT_IMEX_SD_2	Use MRISTep’s default 2nd-order solve-decoupled ImEx method (ARKODE_IMEX_MRI_GARK_TRAPEZOIDAL).
MRISTEP_DEFAULT_IMEX_SD_3	Use MRISTep’s default 3rd-order solve-decoupled ImEx method (ARKODE_IMEX_MRI_GARK3b).

continues on next page

Table 18.1 – continued from previous page

MRISTEP_DEFAULT_IMEX_SD_4	Use MRISTep’s default 4th-order solve-decoupled ImEx method (ARKODE_IMEX_MRI_GARK4).
MRISTEP_DEFAULT_IMEX_SD_2_AD	Use MRISTep’s default 2nd-order solve-decoupled adaptive ImEx method (ARKODE_IMEX_MRI_SR21).
MRISTEP_DEFAULT_IMEX_SD_3_AD	Use MRISTep’s default 3rd-order solve-decoupled adaptive ImEx method (ARKODE_IMEX_MRI_SR32).
MRISTEP_DEFAULT_IMEX_SD_4_AD	Use MRISTep’s default 4th-order solve-decoupled adaptive ImEx method (ARKODE_IMEX_MRI_SR43).

Table 18.2: ARKODE output constants

Shared output constants		
ARK_SUCCESS	0	Successful function return.
ARK_TSTOP_RETURN	1	ARKODE succeeded by reaching the specified stopping point.
ARK_ROOT_RETURN	2	ARKODE succeeded and found one more more roots.
ARK_WARNING	99	ARKODE succeeded but an unusual situation occurred.
ARK_TOO_MUCH_WORK	-1	The solver took <code>mxstep</code> internal steps but could not reach <code>tout</code> .
ARK_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
ARK_ERR_FAILURE	-3	Error test failures occurred too many times during one internal time step, or the minimum step size was reached.
ARK_CONV_FAILURE	-4	Convergence test failures occurred too many times during one internal time step, or the minimum step size was reached.
ARK_LINIT_FAIL	-5	The linear solver’s initialization function failed.
ARK_LSETUP_FAIL	-6	The linear solver’s setup function failed in an unrecoverable manner.
ARK_LSOLVE_FAIL	-7	The linear solver’s solve function failed in an unrecoverable manner.
ARK_RHSFUNC_FAIL	-8	The right-hand side function failed in an unrecoverable manner.
ARK_FIRST_RHSFUNC_ERR	-9	The right-hand side function failed at the first call.
ARK_REPTD_RHSFUNC_ERR	-10	The right-hand side function had repeated recoverable errors.
ARK_UNREC_RHSFUNC_ERR	-11	The right-hand side function had a recoverable error, but no recovery is possible.
ARK_RTFUNC_FAIL	-12	The rootfinding function failed in an unrecoverable manner.
ARK_LFREE_FAIL	-13	The linear solver’s memory deallocation function failed.
ARK_MASSINIT_FAIL	-14	The mass matrix linear solver’s initialization function failed.
ARK_MASSSETUP_FAIL	-15	The mass matrix linear solver’s setup function failed in an unrecoverable manner.
ARK_MASSSOLVE_FAIL	-16	The mass matrix linear solver’s solve function failed in an unrecoverable manner.
ARK_MASSFREE_FAIL	-17	The mass matrix linear solver’s memory deallocation function failed.
ARK_MASSMULT_FAIL	-18	The mass matrix-vector product function failed.
ARK_CONSTR_FAIL	-19	The inequality constraint test failed repeatedly or failed with the minimum step size.
ARK_MEM_FAIL	-20	A memory allocation failed.
ARK_MEM_NULL	-21	The <code>arkode_mem</code> 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 <a href="#">ARKStepCreate()</a> , <a href="#">ERKStepCreate()</a> , or <a href="#">MRISTepCreate()</a> .
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.

continues on next page

Table 18.2 – continued from previous page

ARK_BAD_DKY	-26	The output derivative vector is NULL.
ARK_TOO_CLOSE	-27	The output and initial times are too close to each other.
ARK_VECTOROP_ERR	-28	An error occurred when calling an <i>N_Vector</i> routine.
ARK-NLS_INIT_FAIL	-29	An error occurred when initializing a SUNNonlinSol module.
ARK-NLS_SETUP_FAIL	-30	A non-recoverable error occurred when setting up a SUNNonlinSol module.
ARK-NLS_SETUP_- RECVR	-31	A recoverable error occurred when setting up a SUNNonlinSol module.
ARK-NLS_OP_ERR	-32	An error occurred when calling a set/get routine in a SUNNonlinSol module.
ARK_INNERSTEP_AT- TACH_ERR	-33	An error occurred when attaching the inner stepper module.
ARK_INNERSTEP_FAIL	-34	An error occurred in the inner stepper module.
ARK_OUTERTOINNER_- FAIL	-35	An error occurred in the MRIStep pre inner integrator function.
ARK_INNERTOOUTER_- FAIL	-36	An error occurred in the MRIStep post inner integrator function.
ARK_POSTPROCESS_- STEP_FAIL	-37	An error occurred in a step postprocessing function.
ARK_POSTPROCESS_- STAGE_FAIL	-38	An error occurred in a stage postprocessing function.
ARK_PRESTEPFN_FAIL	-39	An error occurred in a prestep function.
ARK_POSTSTEPFN_FAIL	-40	An error occurred in a poststep function.
ARK_PRERHS_FAIL	-41	An error occurred in a pre-RHS function.
ARK_USER_PREDICT_- FAIL	-42	An error occurred in a user predictor function.
ARK_INTERP_FAIL	-43	An error occurred in the ARKODE polynomial interpolation module.
ARK_INVALID_TABLE	-44	An invalid Butcher or MRI table was encountered.
ARK_CONTEXT_ERR	-45	An error occurred with the SUNDIALS context object
ARK_RELAX_FAIL	-46	An error occurred in computing the relaxation parameter
ARK_RELAX_MEM_FAIL	-47	The relaxation memory structure is NULL
ARK_RELAX_FUNC_FAIL	-48	The relaxation function returned an unrecoverable error
ARK_RELAX_JAC_FAIL	-49	The relaxation Jacobian function returned an unrecoverable error
ARK_CONTROLLER_ERR	-50	An error with a SUNAdaptController object was encountered.
ARK_STEPPER_UNSUP- PORTED	-51	An operation was not supported by the current time-stepping module.
ARK_DOMEIG_FAIL	-52	The dominant eigenvalue function failed. It is either not provided or returns an illegal value.
ARK_MAX_STAGE_- LIMIT_FAIL	-53	Stepper failed to achieve stable results. Either reduce the step size or increase the stage_max_limit
ARK_SUNSTEPPER_ERR	-54	An error occurred in the SUNStepper module.
ARK_STEP_DIRECTION_- ERR	-55	An error occurred changing the step direction.
ARK_ADJ_CHECK- POINT_FAIL	-56	An occurred when checkpointing a state during the adjoint integration.
ARK_ADJ_RECOMPUTE_- FAIL	-57	An occurred recomputing steps during the adjoint integration.
ARK_SUNADJSTEPPER_- ERR	-58	An error occurred in the SUNAdjStepper module.
ARK_DEE_FAIL	-59	An error occurred in the SUNDomEigEstimator module.
ARK_STEP_H0_FAIL	-60	Time stepping module was unable to set the initial step.
ARK_UNRECOGNIZED_- ERROR	-99	An unknown error was encountered.
<b>ARKLS linear solver module output constants</b>		

continues on next page

Table 18.2 – continued from previous page

ARKLS_SUCCESS	0	Successful function return.
ARKLS_MEM_NULL	-1	The <code>arkode_mem</code> argument was NULL.
ARKLS_LMEM_NULL	-2	The ARKLS linear solver interface has not been initialized.
ARKLS_ILL_INPUT	-3	The ARKLS solver interface is not compatible with the current <i>N_Vector</i> module, or an input value was illegal.
ARKLS_MEM_FAIL	-4	A memory allocation request failed.
ARKLS_PMEM_NULL	-5	The preconditioner module has not been initialized.
ARKLS_MASSMEM_- NULL	-6	The ARKLS mass-matrix linear solver interface has not been initialized.
ARKLS_JACFUNC_UN- RECV	-7	The Jacobian function failed in an unrecoverable manner.
ARKLS_JACFUNC_- RECV	-8	The Jacobian function had a recoverable error.
ARKLS_MASSFUNC_UN- RECV	-9	The mass matrix function failed in an unrecoverable manner.
ARKLS_MASSFUNC_- RECV	-10	The mass matrix function had a recoverable error.
ARKLS_SUNMAT_FAIL	-11	An error occurred with the current <i>SUNMatrix</i> module.
ARKLS_SUNLS_FAIL	-12	An error occurred with the current <i>SUNLinearSolver</i> module.

**enum ARKRelaxSolver**

Nonlinear solver identifiers used to specify the method for solving (2.63) when relaxation is enabled.

enumerator **ARK\_RELAX\_NEWTON**

Newton's method

enumerator **ARK\_RELAX\_BRENT**

Brent's method

# Chapter 19

## Butcher Tables

Here we catalog the full set of Butcher tables included in ARKODE. We group these into four categories: *explicit*, *implicit*, *additive* and *symplectic partitioned*. 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):

$c_1$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$c_2$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$c_3$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
$q$	$b_1$	$b_2$	$b_3$
$p$	$\tilde{b}_1$	$\tilde{b}_2$	$\tilde{b}_3$

where here the method and embedding share stage  $A$  and  $c$  values, but use their stages  $z_i$  differently through the coefficients  $b$  and  $\tilde{b}$  to generate methods of orders  $q$  (the main method) and  $p$  (the embedding, typically  $q = p + 1$ , though sometimes this is reversed).

Method authors often use different naming conventions to categorize their methods. For each of the methods below with an embedding, we follow the uniform naming convention:

**NAME-S-P-Q**

where here

- **NAME** is the author or the name provided by the author (if applicable),
- **S** is the number of stages in the method,
- **P** is the global order of accuracy for the embedding,
- **Q** is the global order of accuracy for the method.

For methods without an embedding (e.g., fixed-step methods) **P** is omitted so that methods follow the naming convention **NAME-S-Q**.

For **SPRK** methods, the naming convention is **SPRK-NAME-S-Q**.

In the code, unique integer IDs are defined inside `arkode_butcher_erk.h` and `arkode_butcher_dirk.h` for each method, which may be used by calling routines to specify the desired method. **SPRK** methods are defined inside `arkode_sprk.h`. These names are specified in **fixed width font** at the start of each method’s section below.

Additionally, for each method we provide a plot of the linear stability region in the complex plane. These have been computed via the following approach. For any Runge–Kutta method as defined above, we may define the stability

function

$$R(\eta) = 1 + \eta b[I - \eta A]^{-1}e,$$

where  $e \in \mathbb{R}^s$  is a column vector of all ones,  $\eta = h\lambda$  and  $h$  is the time step size. If the stability function satisfies  $|R(\eta)| \leq 1$  for all eigenvalues,  $\lambda$ , of  $\frac{\partial}{\partial y}f(t, y)$  for a given IVP, then the method will be linearly stable for that problem and step size. The stability region

$$S = \{\eta \in \mathbb{C} : |R(\eta)| \leq 1\}$$

is typically given by an enclosed region of the complex plane, so it is standard to search for the border of that region in order to understand the method. Since all complex numbers with unit magnitude may be written as  $e^{i\theta}$  for some value of  $\theta$ , 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  $16\pi$ . Since some angles will correspond to multiple locations on the stability boundary, by going beyond  $2\pi$  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  $\theta \in \mathbf{Theta}$ , 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  $\theta$ , starting with an initial-initial guess of  $\eta = 0$  for  $\theta = 0$ .

3. We then plot the resulting  $\eta$  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  $\eta_0$ . Resultingly, methods whose linear stability boundary is located entirely in the right half-plane indicate an *A-stable* method.

## 19.1 Explicit Butcher tables

In the category of explicit Runge–Kutta methods, ARKODE includes methods that have orders 2 through 9, with embeddings that are of orders 1 through 8. ARKODE's explicit Butcher tables are provided in the enumeration

enum **ARKODE\_ERKTableID**

with values specified in [Table 19.1](#).



Table 19.1: Explicit Butcher tables. The default method for each order is marked with an asterisk (\*).

Method ID	Stages	Embedded Order	Order
<a href="#">ARKODE_FORWARD_EULER_1_1</a>	1	—	1*
<a href="#">ARKODE_RALSTON_3_1_2</a>	3	1	2*
<a href="#">ARKODE_HEUN_EULER_2_1_2</a>	2	1	2
<a href="#">ARKODE_RALSTON_EULER_2_1_2</a>	2	1	2
<a href="#">ARKODE_EXPLICIT_MIDPOINT_EULER_2_1_2</a>	2	1	2
<a href="#">ARKODE_ARK2_ERK_3_1_2</a>	3	1	2
<a href="#">ARKODE_BOGACKI_SHAMPINE_4_2_3</a>	4	2	3*
<a href="#">ARKODE_ARK324L2SA_ERK_4_2_3</a>	4	2	3
<a href="#">ARKODE_SHU_OSHER_3_2_3</a>	3	2	3
<a href="#">ARKODE_KNOTH_WOLKE_3_3</a>	3	—	3
<a href="#">ARKODE_SOFRONIOU_SPALETTA_5_3_4</a>	5	3	4*
<a href="#">ARKODE_ZONNEVELD_5_3_4</a>	5	3	4
<a href="#">ARKODE_ARK436L2SA_ERK_6_3_4</a>	6	3	4
<a href="#">ARKODE_ARK437L2SA_ERK_7_3_4</a>	7	3	4
<a href="#">ARKODE_SAYFY_ABURUB_6_3_4</a>	6	3	4
<a href="#">ARKODE_TSITOURAS_7_4_5</a>	7	4	5*
<a href="#">ARKODE_CASH_KARP_6_4_5</a>	6	4	5
<a href="#">ARKODE_FEHLBERG_6_4_5</a>	6	4	5
<a href="#">ARKODE_DORMAND_PRINCE_7_4_5</a>	7	4	5
<a href="#">ARKODE_ARK548L2SA_ERK_8_4_5</a>	8	4	5
<a href="#">ARKODE_ARK548L2SAb_ERK_8_4_5</a>	8	4	5
<a href="#">ARKODE_VERNER_9_5_6</a>	9	5	6*
<a href="#">ARKODE_VERNER_8_5_6</a>	8	5	6
<a href="#">ARKODE_VERNER_10_6_7</a>	10	6	7*
<a href="#">ARKODE_VERNER_13_7_8</a>	13	7	8*
<a href="#">ARKODE_FEHLBERG_13_7_8</a>	13	7	8
<a href="#">ARKODE_VERNER_16_8_9</a>	16	8	9*

enumerator **ARKODE\_FORWARD\_EULER\_1\_1**

Accessible via the constant `ARKODE_FORWARD_EULER_1_1` to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_FORWARD\_EULER\_1\_1" to [ARKStepSetTableName\(\)](#), [ERKStepSetTableName\(\)](#) or [ARKodeButcherTable\\_LoadERKByName\(\)](#). This is the default 1st order explicit method (from [40]).

$$\begin{array}{c|c} 0 & 0 \\ \hline 1 & 1 \end{array}$$

enumerator **ARKODE\_RALSTON\_3\_1\_2**

Accessible via the constant `ARKODE_RALSTON_3_1_2` to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_RALSTON\_3\_1\_2" to [ARKStepSetTableName\(\)](#), [ERKStepSetTableName\(\)](#) or [ARKodeButcherTable\\_LoadERKByName\(\)](#). This is the default 2nd order explicit method (primary method from [84]).

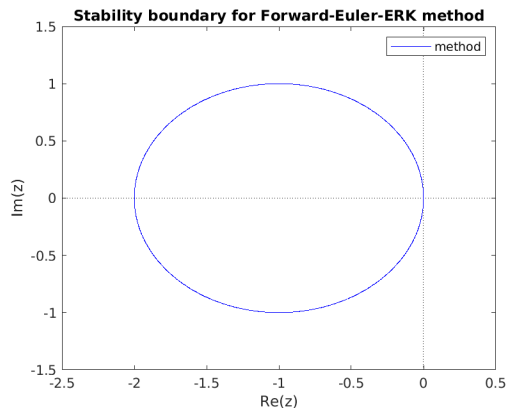


Fig. 19.1: Linear stability region for the forward Euler method.

Changed in version 6.3.0: Added as the default 2nd order explicit method

0	0	0	0
$\frac{2}{3}$	$\frac{2}{3}$	0	0
1	$\frac{1}{4}$	$\frac{3}{4}$	0
2	$\frac{1}{4}$	$\frac{3}{4}$	0
1	$\frac{5}{37}$	$\frac{2}{3}$	$\frac{22}{111}$

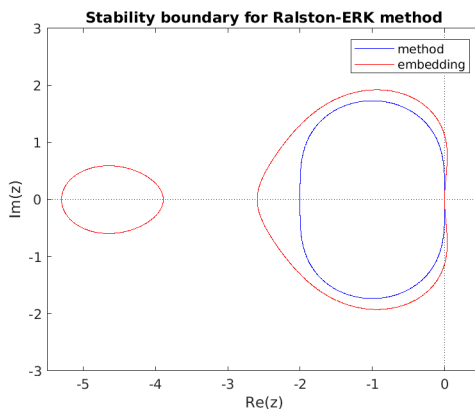


Fig. 19.2: Linear stability region for the Ralston method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_HEUN\_EULER\_2\_1\_2**

Accessible via the constant `ARKODE_HEUN_EULER_2_1_2` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_HEUN\_EULER\_2\_1\_2" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. (primary method from [89]).

Changed in version 6.3.0: Replaced by `ARKODE_RALSTON_3_1_2` as the default 2nd order explicit method

0	0	0
1	1	0
2	$\frac{1}{2}$	$\frac{1}{2}$
1	1	0

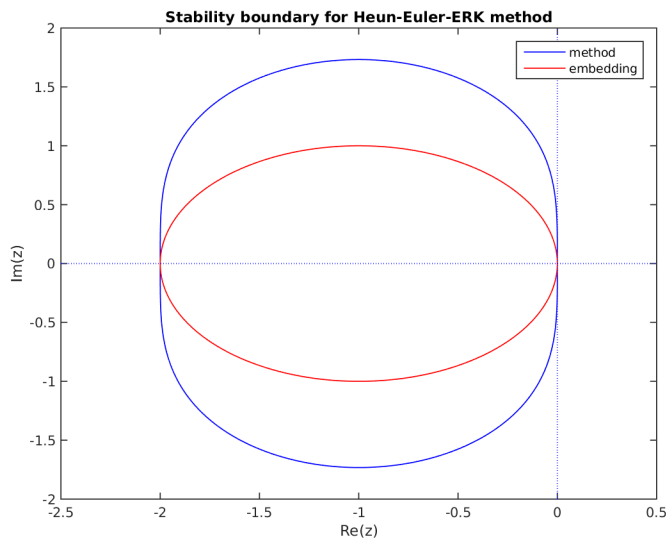


Fig. 19.3: Linear stability region for the Heun-Euler method. The method's region is outlined in blue; the embedding's region is in red.

enumerator `ARKODE_RALSTON_EULER_2_1_2`

Accessible via the constant `ARKODE_RALSTON_EULER_2_1_2` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_RALSTON\_EULER\_2\_1\_2" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. (primary method from [84]).

0	0	0
$\frac{2}{3}$	$\frac{2}{3}$	0
2	$\frac{1}{4}$	$\frac{3}{4}$
1	1	0

enumerator `ARKODE_EXPLICIT_MIDPOINT_EULER_2_1_2`

Accessible via the constant `ARKODE_EXPLICIT_MIDPOINT_EULER_2_1_2` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_EXPLICIT\_MIDPOINT\_EULER\_2\_1\_2" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_Load-`

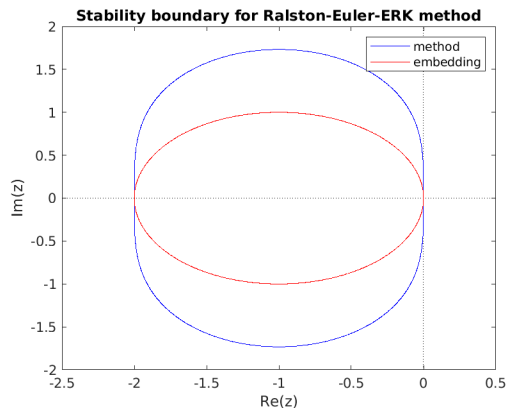


Fig. 19.4: Linear stability region for the Ralston-Euler method. The method's region is outlined in blue; the embedding's region is in red.

*ERKByName()*. (primary method from [89]).

0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0
2	0	1
1	1	0

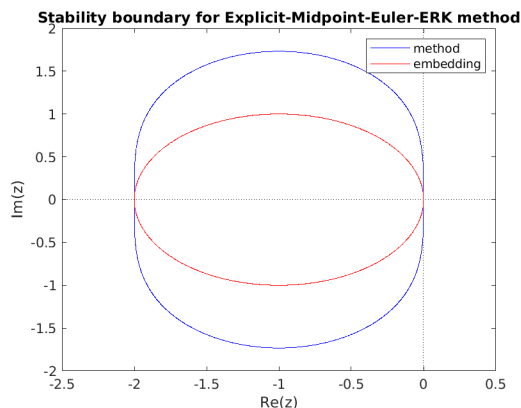


Fig. 19.5: Linear stability region for the Explicit-Midpoint-Euler method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ARK2\_ERK\_3\_1\_2**

Accessible via the constant **ARKODE\_ARK2\_ERK\_3\_1\_2** to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable\_LoadERK()*. Accessible via the string "ARKODE\_ARK2\_ERK\_3\_1\_2" to *ARKStepSetTableName()*, *ERKStepSetTableName()* or *ARKodeButcherTable\_LoadERKByName()*. This is the explicit portion of

the default 2nd order additive method (the explicit portion of the ARK2 method from [50]).

0	0	0	0
$2 - \sqrt{2}$	$2 - \sqrt{2}$	0	0
1	$1 - \frac{3+2\sqrt{2}}{6}$	$\frac{3+2\sqrt{2}}{6}$	0
2	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}$	$1 - \frac{1}{\sqrt{2}}$
1	$\frac{4-\sqrt{2}}{8}$	$\frac{4-\sqrt{2}}{8}$	$\frac{1}{2\sqrt{2}}$

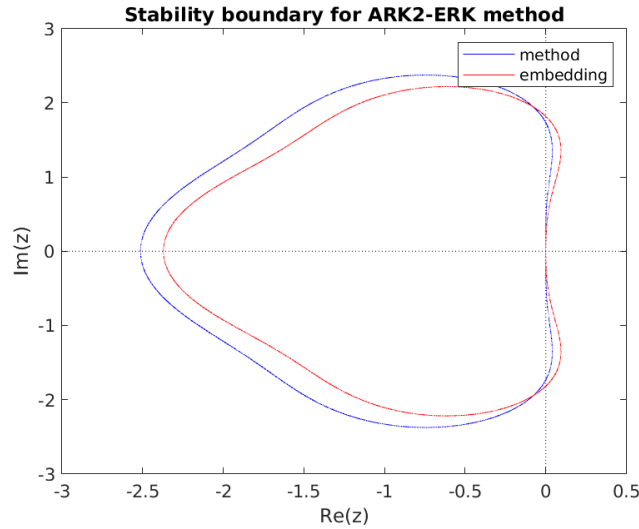


Fig. 19.6: Linear stability region for the ARK2-ERK method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_BOGACKI\_SHAMPINE\_4\_2\_3**

Accessible via the constant `ARKODE_BOGACKI_SHAMPINE_4_2_3` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_BOGACKI\_SHAMPINE\_4\_2\_3" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the default 3rd order explicit method (from [20]).

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
3	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
2	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

enumerator **ARKODE\_ARK324L2SA\_ERK\_4\_2\_3**

Accessible via the constant `ARKODE_ARK324L2SA_ERK_4_2_3` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_ARK324L2SA\_ERK\_4\_2\_3"

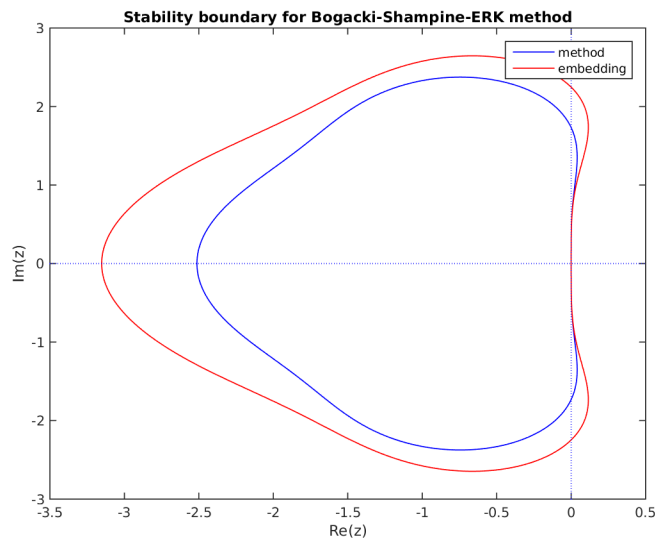


Fig. 19.7: Linear stability region for the Bogacki-Shampine method. The method's region is outlined in blue; the embedding's region is in red.

to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the explicit portion of the default 3rd order additive method (the explicit portion of the ARK3(2)4L[2]SA method from [68]).

0	0	0	0	0
$\frac{1767732205903}{2027836641118}$	$\frac{1767732205903}{2027836641118}$	0	0	0
$\frac{3}{5}$	$\frac{5535828885825}{10492691773637}$	$\frac{788022342437}{10882634858940}$	0	0
1	$\frac{6485989280629}{16251701735622}$	$\frac{4246266847089}{9704473918619}$	$\frac{10755448449292}{10357097424841}$	0
3	$\frac{1471266399579}{7840856788654}$	$\frac{4482444167858}{7529755066697}$	$\frac{11266239266428}{11593286722821}$	$\frac{1767732205903}{4055673282236}$
2	$\frac{2756255671327}{12835298489170}$	$\frac{10771552573575}{22201958757719}$	$\frac{9247589265047}{10645013368117}$	$\frac{2193209047091}{5459859503100}$

enumerator `ARKODE_SHU_OSHER_3_2_3`

Accessible via the constant `ARKODE_SHU_OSHER_3_2_3` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_SHU\_OSHER\_3\_2\_3" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. (from [101] with embedding from [45]).

0	0	0	0
1	1	0	0
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	0
3	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{2}{3}$
2	$\frac{291485418878409}{1000000000000000}$	$\frac{291485418878409}{1000000000000000}$	$\frac{208514581121591}{5000000000000000}$

enumerator `ARKODE_KNOTH_WOLKE_3_3`

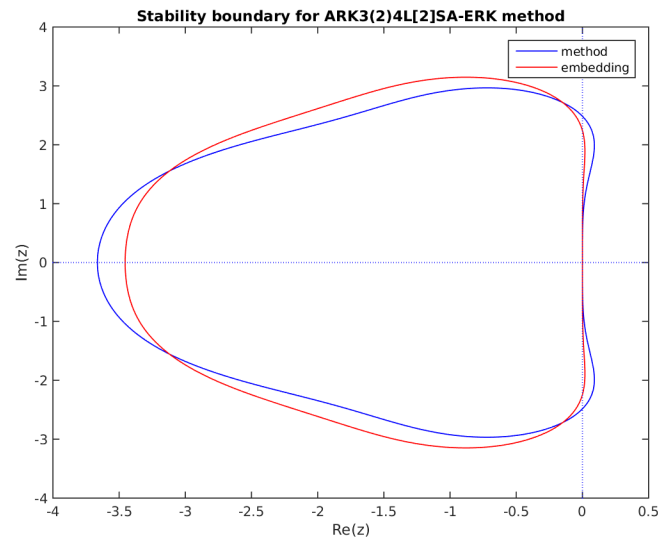


Fig. 19.8: Linear stability region for the explicit ARK-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

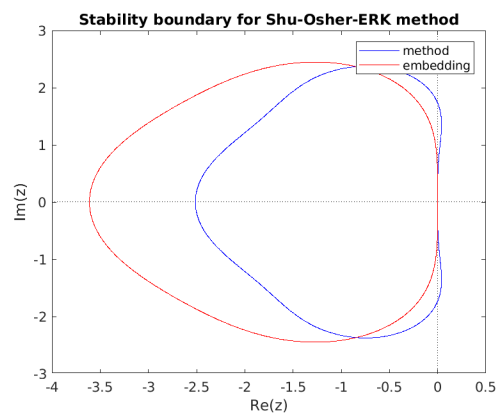


Fig. 19.9: Linear stability region for the Shu-Osher method. The method's region is outlined in blue; the embedding's region is in red.

Accessible via the constant `ARKODE_KNOTH_WOLKE_3_3` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()`, or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_KNOTH\_WOLKE\_3\_3" to `ARKStepSetTableName()`, `ERKStepSetTableName()`, or `ARKodeButcherTable_LoadERKByName()`. This is the default 3th order slow and fast MRISep method (from [74]).

0	0	0	0
$\frac{1}{3}$	$\frac{1}{3}$	0	0
$\frac{3}{4}$	$-\frac{3}{16}$	$\frac{15}{16}$	0
3	$\frac{1}{6}$	$\frac{3}{10}$	$\frac{8}{15}$

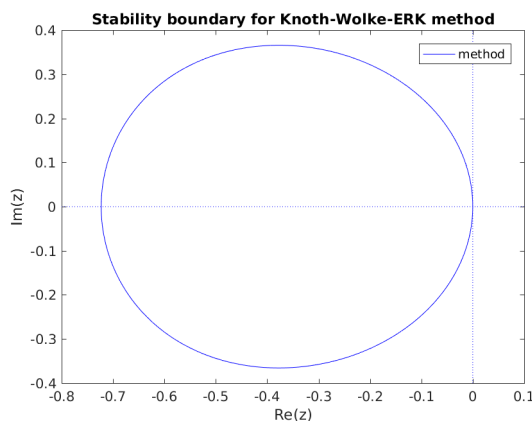


Fig. 19.10: Linear stability region for the Knoth-Wolke method

enumerator `ARKODE_SOFRONIOU_SPALETTA_5_3_4`

Accessible via the constant `ARKODE_SOFRONIOU_SPALETTA_5_3_4` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_SOFRONIOU\_SPALETTA\_5\_3\_4" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the default 4th order explicit method. (from [106]).

Changed in version 6.3.0: Made the default 4th order explicit method

0	0	0	0	0	0
$\frac{2}{5}$	$\frac{2}{5}$	0	0	0	0
$\frac{3}{5}$	$-\frac{3}{20}$	$\frac{3}{4}$	0	0	0
1	$\frac{19}{44}$	$-\frac{15}{44}$	$\frac{10}{11}$	0	0
1	$\frac{11}{72}$	$\frac{25}{72}$	$\frac{25}{72}$	$\frac{11}{72}$	0
4	$\frac{11}{72}$	$\frac{25}{72}$	$\frac{25}{72}$	$\frac{11}{72}$	0
3	$\frac{1251515}{8970912}$	$\frac{3710105}{8970912}$	$\frac{2519695}{8970912}$	$\frac{61105}{8970912}$	$\frac{119041}{747576}$

enumerator `ARKODE_ZONNEVELD_5_3_4`

Accessible via the constant `ARKODE_ZONNEVELD_5_3_4` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()`, or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_ZONNEVELD\_5\_3\_4" to `ARKStepSetTableName()`, `ERKStepSetTableName()`, or `ARKodeButcherTable_LoadERKByName()`. (from [127]).



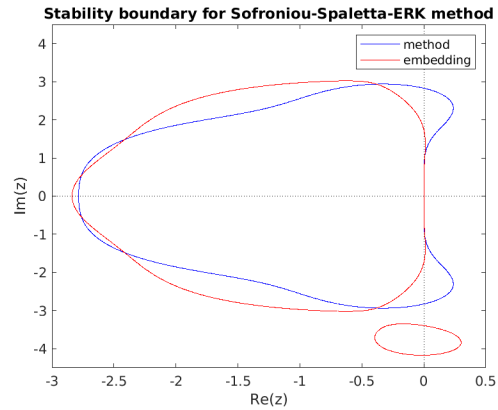


Fig. 19.11: Linear stability region for the Sofroniou-Spaletta method. The method's region is outlined in blue; the embedding's region is in red.

Changed in version 6.3.0: Replaced by ARKODE\_SOFRONIOU\_SPALETTA\_5\_3\_4 as the default 4th order explicit method

0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0
1	0	0	1	0	0
$\frac{3}{4}$	$\frac{5}{32}$	$\frac{7}{32}$	$\frac{13}{32}$	$-\frac{1}{32}$	0
4	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$	0
3	$-\frac{1}{2}$	$\frac{7}{3}$	$\frac{7}{3}$	$\frac{13}{6}$	$-\frac{16}{3}$

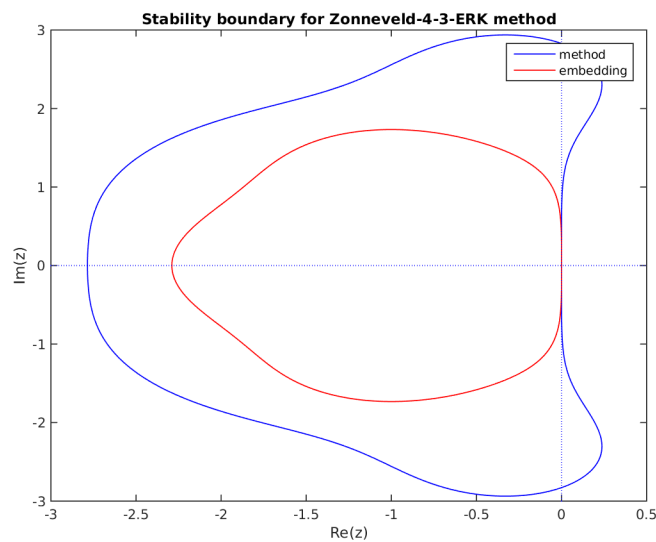


Fig. 19.12: Linear stability region for the Zonneveld method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ARK436L2SA\_ERK\_6\_3\_4**

Accessible via the constant `ARKODE_ARK436L2SA_ERK_6_3_4` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_ARK436L2SA\_ERK\_6\_3\_4" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the explicit portion of the ARK4(3)6L[2]SA method from [68].

Changed in version 6.3.0: Replaced by `ARKODE_ARK437L2SA_ERK_7_3_4` as the explicit portion of the default 4th order additive method

0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	0
$\frac{83}{250}$	$\frac{13861}{62500}$	$\frac{6889}{62500}$	0	0	0	0
$\frac{31}{50}$	$-\frac{116923316275}{2393684061468}$	$-\frac{2731218467317}{15368042101831}$	$\frac{9408046702089}{11113171139209}$	0	0	0
$\frac{17}{20}$	$-\frac{451086348788}{2902428689909}$	$-\frac{2682348792572}{7519795681897}$	$\frac{12662868775082}{11960479115383}$	$\frac{3355817975965}{11060851509271}$	0	0
1	$\frac{647845179188}{3216320057751}$	$\frac{73281519250}{8382639484533}$	$\frac{552539513391}{3454668386233}$	$\frac{3354512671639}{8306763924573}$	$\frac{4040}{17871}$	0
4	$\frac{82889}{524892}$	0	$\frac{15625}{83664}$	$\frac{69875}{102672}$	$-\frac{2260}{8211}$	$\frac{1}{4}$
3	$\frac{4586570599}{29645900160}$	0	$\frac{178811875}{945068544}$	$\frac{814220225}{1159782912}$	$-\frac{3700637}{11593932}$	$\frac{61727}{225920}$

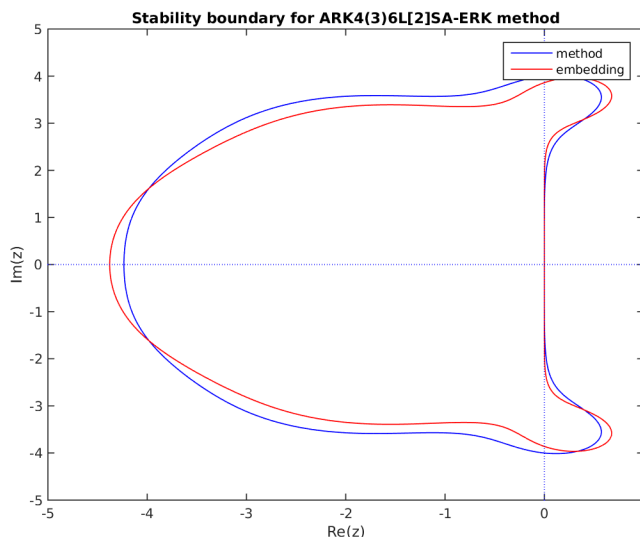


Fig. 19.13: Linear stability region for the ARK436L2SA-ERK-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ARK437L2SA\_ERK\_7\_3\_4**

Accessible via the constant `ARKODE_ARK437L2SA_ERK_7_3_4` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_ARK437L2SA\_ERK\_7\_3\_4" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the explicit portion of the default 4th order additive method and the explicit portion of the ARK4(3)7L[2]SA method from [71].

Changed in version 6.3.0: Made the explicit portion of the default 4th order additive method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

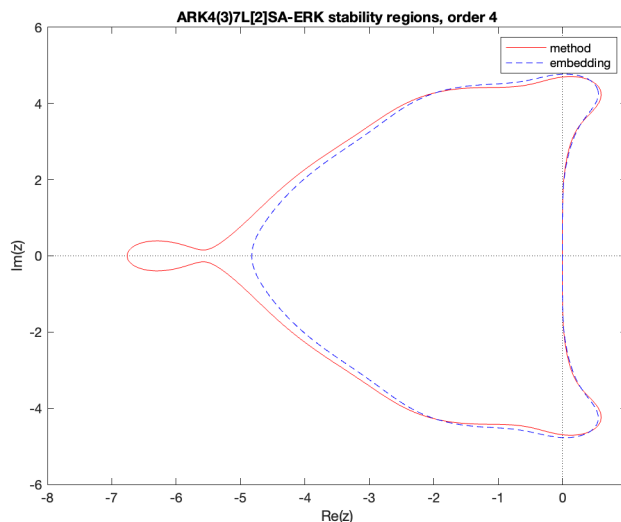


Fig. 19.14: Linear stability region for the ARK437L2SA-ERK-7-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_SAYFY\_ABURUB\_6\_3\_4**

Accessible via the constant **ARKODE\_SAYFY\_ABURUB\_6\_3\_4** to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_SAYFY\_ABURUB\_6\_3\_4" to [ARKStepSetTableName\(\)](#), [ERKStepSetTableName\(\)](#) or [ARKodeButcherTable\\_LoadERKByName\(\)](#). (from [95]).

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
$\frac{1}{2}$	$\frac{137}{1000}$	$\frac{113}{500}$	$\frac{137}{1000}$	0	0	0
1	$\frac{113}{250}$	$-\frac{113}{125}$	$-\frac{137}{250}$	0	2	0
4	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{12}$	0	$\frac{1}{3}$	$\frac{1}{12}$
3	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	0	0	0

enumerator **ARKODE\_TSITOURAS\_7\_4\_5**

Accessible via the constant **ARKODE\_TSITOURAS\_7\_4\_5** to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_TSITOURAS\_7\_4\_5" to [ARKStepSetTableName\(\)](#), [ERKStepSetTableName\(\)](#) or [ARKodeButcherTable\\_LoadERKByName\(\)](#). This is the default 5th order explicit method (from [120]).

Changed in version 6.3.0: Added as the default 5th order explicit method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

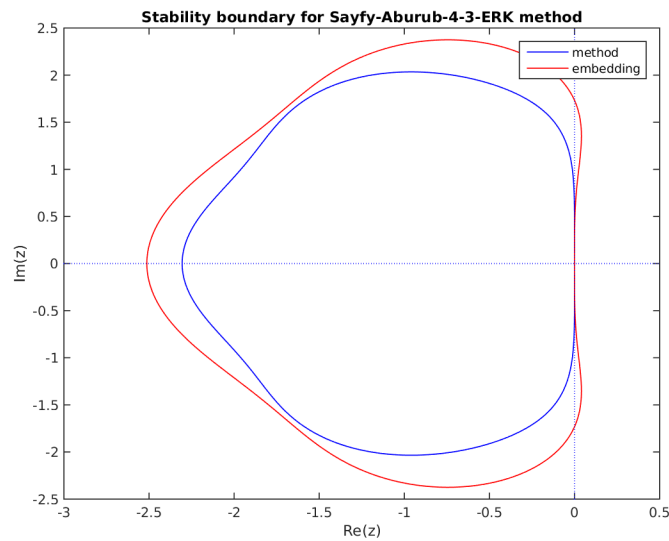


Fig. 19.15: Linear stability region for the Sayfy-Aburub-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

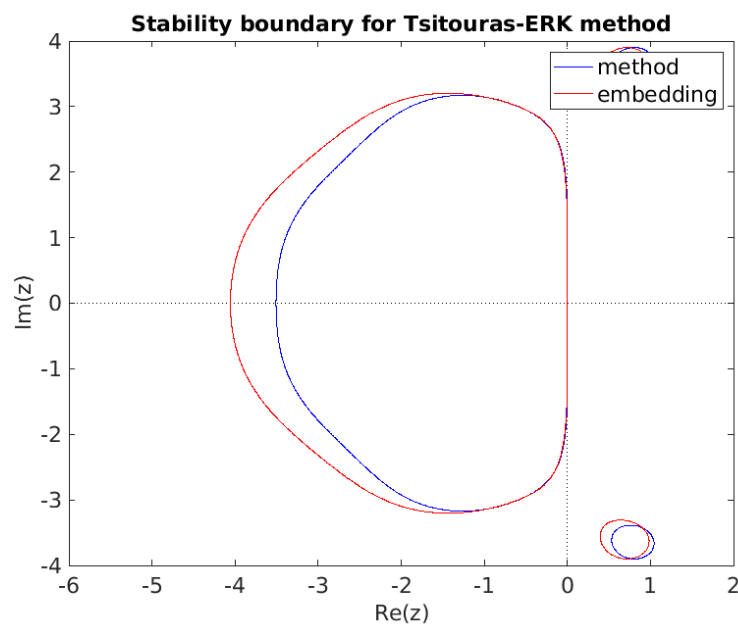


Fig. 19.16: Linear stability region for the Tsitouras method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_CASH\_KARP\_6\_4\_5**

Accessible via the constant **ARKODE\_CASH\_KARP\_6\_4\_5** to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_CASH\_KARP\_6\_4\_5" to [ARKStepSetTableName\(\)](#), [ERKStepSetTableName\(\)](#) or [ARKodeButcherTable\\_LoadERKByName\(\)](#). (from [29]).

Changed in version 6.3.0: Replaced by **ARKODE\_TSITOURAS\_7\_4\_5** as the default 5th order explicit method

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
5	$\frac{37}{378}$	0	$\frac{250}{621}$	$\frac{125}{594}$	0	$\frac{512}{1771}$
4	$\frac{2825}{27648}$	0	$\frac{18575}{48384}$	$\frac{13525}{55296}$	$\frac{277}{14336}$	$\frac{1}{4}$

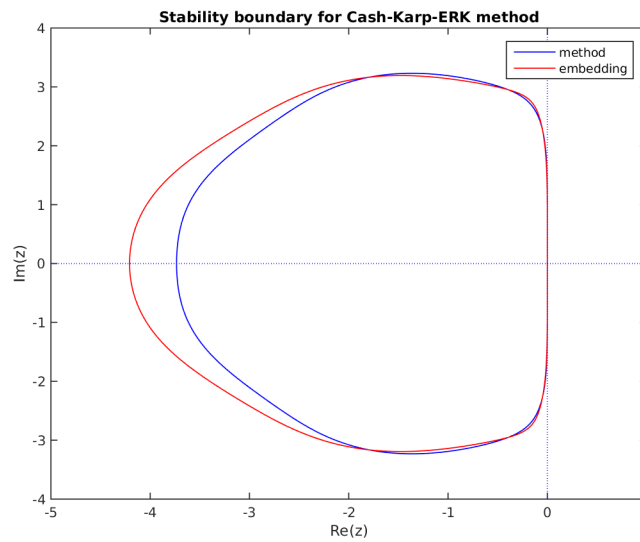


Fig. 19.17: Linear stability region for the Cash-Karp method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_FEHLBERG\_6\_4\_5**

Accessible via the constant **ARKODE\_FEHLBERG\_6\_4\_5** to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_FEHLBERG\_6\_4\_5" to [ARKStepSetTable-](#)

*Name()*, *ERKStepSetTableName()* or *ARKodeButcherTable\_LoadERKByName()*. (from [43]).

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
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

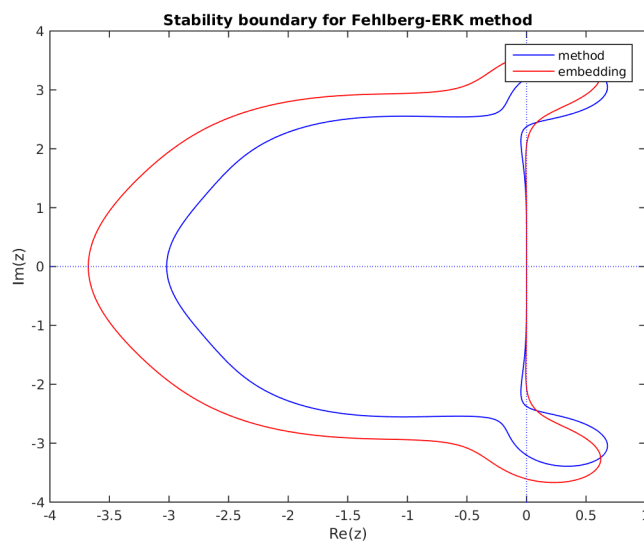


Fig. 19.18: Linear stability region for the Fehberg method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_DORMAND\_PRINCE\_7\_4\_5**

Accessible via the constant **ARKODE\_DORMAND\_PRINCE\_7\_4\_5** to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable\_LoadERK()*. Accessible via the string "ARKODE\_DORMAND\_PRINCE\_7\_4\_5" to *ARKStepSetTableName()*, *ERKStepSetTableName()* or *ARKodeButcherTable\_LoadERKByName()*. (from

[37]).

0	0	0	0	0	0	0	0
$\frac{1}{5}$	$\frac{1}{5}$	0	0	0	0	0	0
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	0	0	0	0	0
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$	0	0	0	0
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$	0	0	0
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$	0	0
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
5	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
4	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

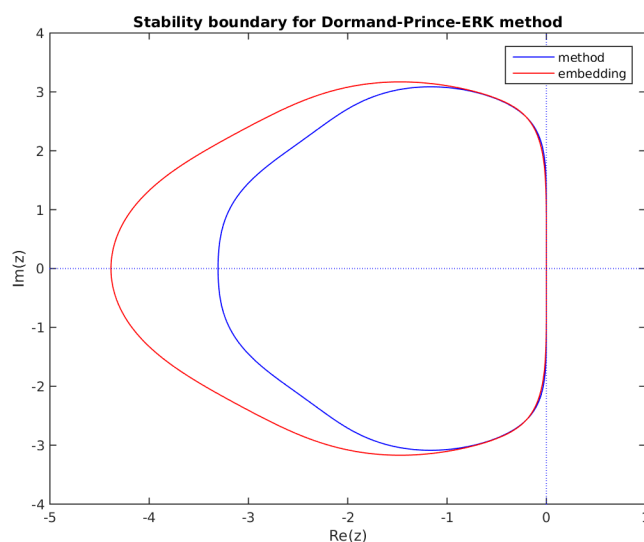


Fig. 19.19: Linear stability region for the Dormand-Prince method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ARK548L2SA\_ERK\_8\_4\_5**

Accessible via the constant **ARKODE\_ARK548L2SA\_ERK\_8\_4\_5** to [`ARKStepSetTableNum\(\)`](#), [`ERKStepSetTableNum\(\)`](#) or [`ARKodeButcherTable\_LoadERK\(\)`](#). Accessible via the string "ARKODE\_ARK548L2SA\_ERK\_8\_4\_5" to [`ARKStepSetTableName\(\)`](#), [`ERKStepSetTableName\(\)`](#) or [`ARKodeButcherTable\_LoadERKByName\(\)`](#). This is the explicit portion of the ARK5(4)8L[2]SA method from [68].

Changed in version 6.3.0: Replaced by **ARKODE\_ARK548L2SAb\_ERK\_8\_4\_5** as the explicit portion of the default 5th order additive method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator **ARKODE\_ARK548L2SAb\_ERK\_8\_4\_5**

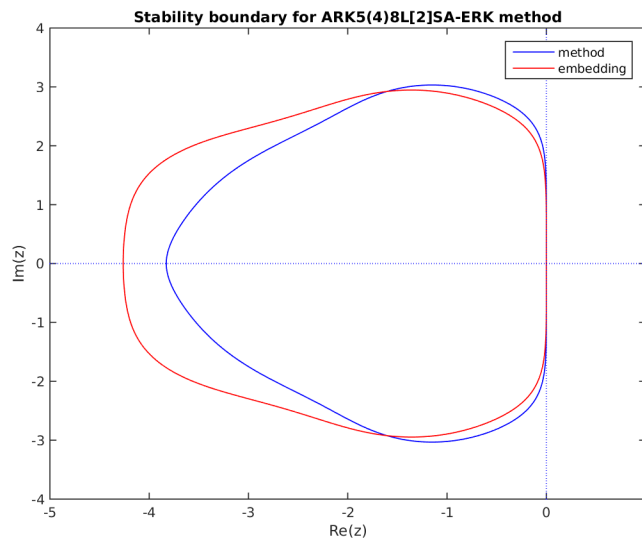


Fig. 19.20: Linear stability region for the explicit ARK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

Accessible via the constant `ARKODE_ARK548L2SAb_ERK_8_4_5` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_ARK548L2SAb\_ERK\_8\_4\_5" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the explicit portion of the default 5th order additive method and the explicit portion of the 5th order ARK5(4)8L[2]SA method from [71].

Changed in version 6.3.0: Made the explicit portion of the default 5th order additive method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator `ARKODE_VERNER_9_5_6`

Accessible via the constant `ARKODE_VERNER_9_5_6` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_VERNER\_9\_5\_6" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the default 6th order explicit method (method IIIxB-6(5) from [121]).

Changed in version 6.3.0: Made the default 6th order explicit method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator `ARKODE_VERNER_8_5_6`

Accessible via the constant `ARKODE_VERNER_8_5_6` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_VERNER\_8\_5\_6" to `ARKStepSetTableName()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. (from [63]).



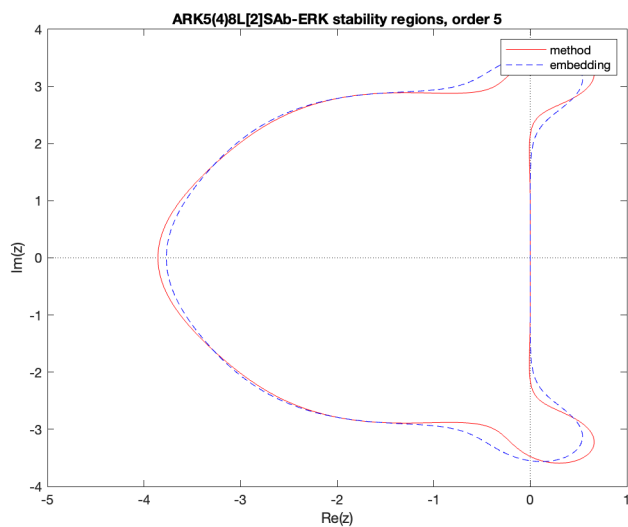


Fig. 19.21: Linear stability region for the ARK548L2SAb-ERK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

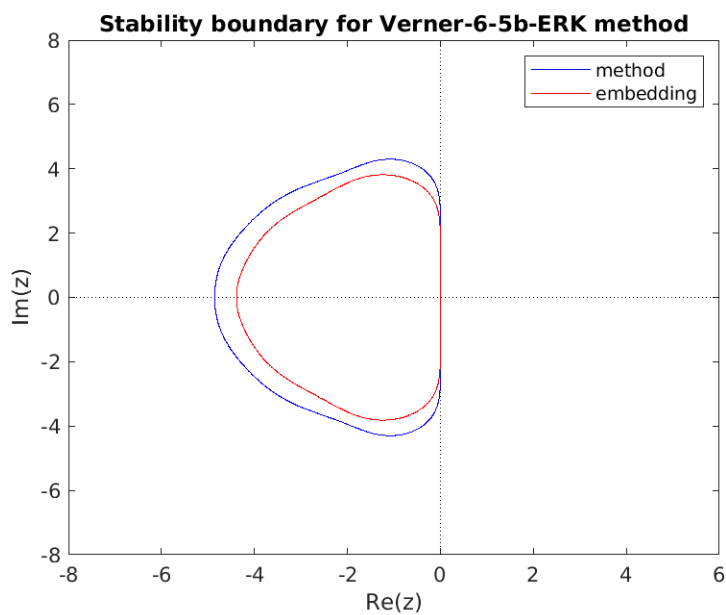


Fig. 19.22: Linear stability region for the Verner-9-5-6 method. The method's region is outlined in blue; the embedding's region is in red.

Changed in version 6.3.0: Replaced by ARKODE\_VERNER\_9\_5\_6 as the default 6th order explicit method

0	0	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{1}{6}$	0	0	0	0	0	0	0
$\frac{4}{15}$	$\frac{4}{75}$	$\frac{16}{75}$	0	0	0	0	0	0
$\frac{2}{3}$	$\frac{5}{6}$	$-\frac{8}{3}$	$\frac{5}{2}$	0	0	0	0	0
$\frac{5}{6}$	$-\frac{165}{64}$	$\frac{55}{6}$	$-\frac{425}{64}$	$\frac{85}{96}$	0	0	0	0
1	$\frac{12}{5}$	-8	$\frac{4015}{612}$	$-\frac{11}{36}$	$\frac{88}{255}$	0	0	0
$\frac{1}{15}$	$-\frac{8263}{15000}$	$\frac{124}{75}$	$-\frac{643}{680}$	$-\frac{81}{250}$	$\frac{2484}{10625}$	0	0	0
1	$\frac{3501}{1720}$	$-\frac{300}{43}$	$\frac{297275}{52632}$	$-\frac{319}{2322}$	$\frac{24068}{84065}$	0	$\frac{3850}{26703}$	0
6	$\frac{3}{40}$	0	$\frac{875}{2244}$	$\frac{23}{72}$	$\frac{264}{1955}$	0	$\frac{125}{11592}$	$\frac{43}{616}$
5	$\frac{13}{160}$	0	$\frac{2375}{5984}$	$\frac{5}{16}$	$\frac{12}{85}$	$\frac{3}{44}$	0	0

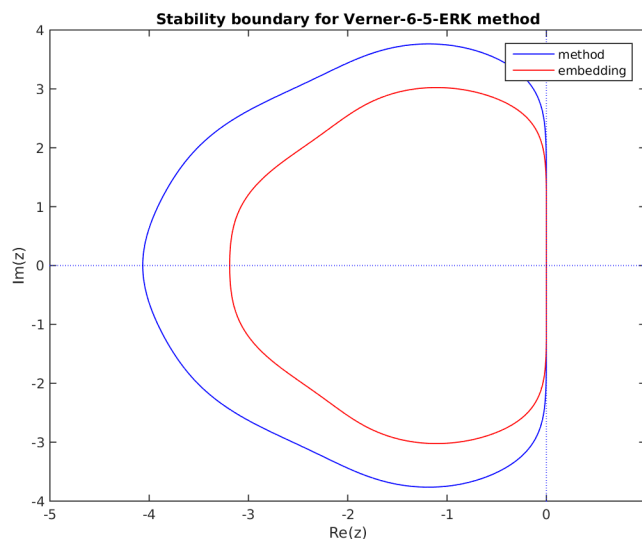


Fig. 19.23: Linear stability region for the Verner-8-5-6 method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_VERNER\_10\_6\_7**

Accessible via the constant ARKODE\_VERNER\_10\_6\_7 to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_VERNER\_10\_6\_7" to [ARKStepSetTableName\(\)](#), [ERKStepSetTableName\(\)](#) or [ARKodeButcherTable\\_LoadERKByName\(\)](#). This is the default 7th order explicit method (from [121]).

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator **ARKODE\_VERNER\_13\_7\_8**

Accessible via the constant ARKODE\_VERNER\_13\_7\_8 to [ARKStepSetTableNum\(\)](#), [ERKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadERK\(\)](#). Accessible via the string "ARKODE\_VERNER\_13\_7\_8" to [ARKStepSetTable-](#)

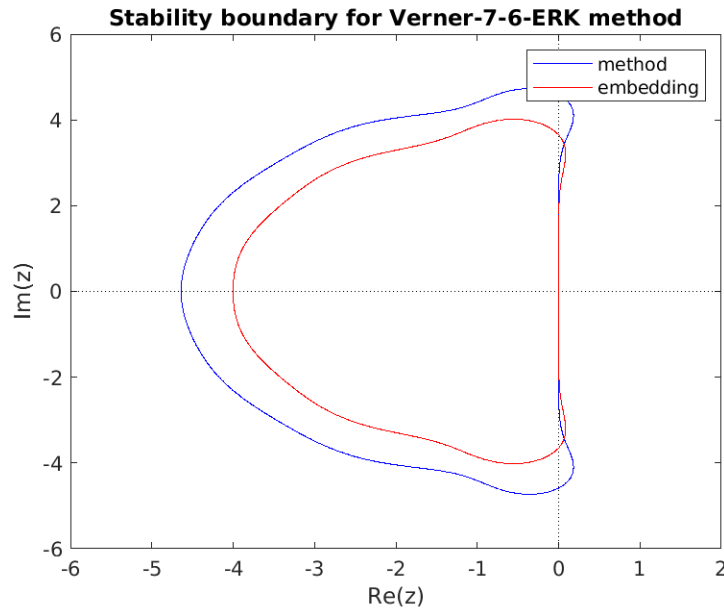


Fig. 19.24: Linear stability region for the Verner-10-6-7 method. The method's region is outlined in blue; the embedding's region is in red.

[\*Name\(\)\*](#), [\*ERKStepSetTableName\(\)\*](#) or [\*ARKodeButcherTable\\_LoadERKByName\(\)\*](#). This is the default 8th order explicit method (method IIIX-8(7) from [121]).

Changed in version 6.3.0: Made the default 8th order explicit method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator **ARKODE\_FEHLBERG\_13\_7\_8**

Accessible via the constant **ARKODE\_FEHLBERG\_13\_7\_8** to [\*ARKStepSetTableNum\(\)\*](#), [\*ERKStepSetTableNum\(\)\*](#) or [\*ARKodeButcherTable\\_LoadERK\(\)\*](#). Accessible via the string "ARKODE\_FEHLBERG\_13\_7\_8" to [\*ARKStepSetTableName\(\)\*](#), [\*ERKStepSetTableName\(\)\*](#) or [\*ARKodeButcherTable\\_LoadERKByName\(\)\*](#). (from [25]).

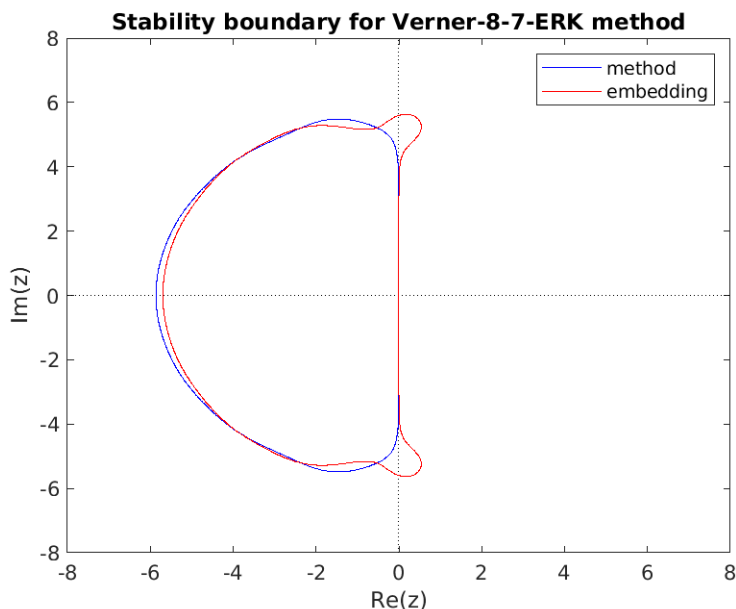


Fig. 19.25: Linear stability region for the Verner-13-7-8 method. The method's region is outlined in blue; the embedding's region is in red.

Changed in version 6.3.0: Replaced by `ARKODE_VERNER_13_7_8` as the default 8th order explicit method

0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\frac{2}{27}$	$\frac{2}{27}$	0	0	0	0	0	0	0	0	0	0	0	0
$\frac{1}{9}$	$\frac{1}{36}$	$\frac{1}{12}$	0	0	0	0	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{1}{24}$	0	$\frac{1}{8}$	0	0	0	0	0	0	0	0	0	0
$\frac{5}{12}$	$\frac{5}{12}$	0	$-\frac{25}{16}$	$\frac{25}{16}$	0	0	0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{20}$	0	0	$\frac{1}{4}$	$\frac{1}{5}$	0	0	0	0	0	0	0	0
$\frac{5}{6}$	$-\frac{25}{108}$	0	0	$\frac{125}{108}$	$-\frac{65}{27}$	$\frac{125}{54}$	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{31}{300}$	0	0	0	$\frac{61}{225}$	$-\frac{2}{9}$	$\frac{13}{900}$	0	0	0	0	0	0
$\frac{2}{3}$	2	0	0	$-\frac{53}{6}$	$\frac{704}{45}$	$-\frac{107}{9}$	$\frac{67}{90}$	3	0	0	0	0	0
$\frac{1}{3}$	$-\frac{91}{108}$	0	0	$\frac{23}{108}$	$-\frac{976}{135}$	$\frac{311}{54}$	$-\frac{19}{60}$	$\frac{17}{6}$	$-\frac{1}{12}$	0	0	0	0
1	$\frac{2383}{4100}$	0	0	$-\frac{341}{164}$	$\frac{4496}{1025}$	$-\frac{301}{82}$	$\frac{2133}{4100}$	$\frac{45}{82}$	$\frac{45}{164}$	$\frac{18}{41}$	0	0	0
0	$\frac{3}{205}$	0	0	0	0	$-\frac{6}{41}$	$-\frac{3}{205}$	$-\frac{3}{41}$	$\frac{3}{41}$	$\frac{6}{41}$	0	0	0
1	$-\frac{1777}{4100}$	0	0	$-\frac{341}{164}$	$\frac{4496}{1025}$	$-\frac{289}{82}$	$\frac{2193}{4100}$	$\frac{51}{82}$	$\frac{33}{164}$	$\frac{12}{41}$	0	1	0
8	0	0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	0	$\frac{41}{840}$	$\frac{41}{840}$
7	$\frac{41}{840}$	0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	$\frac{41}{840}$	0	0

enumerator `ARKODE_VERNER_16_8_9`

Accessible via the constant `ARKODE_VERNER_16_8_9` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. Accessible via the string "ARKODE\_VERNER\_16\_8\_9" to `ARKStepSetTable-`

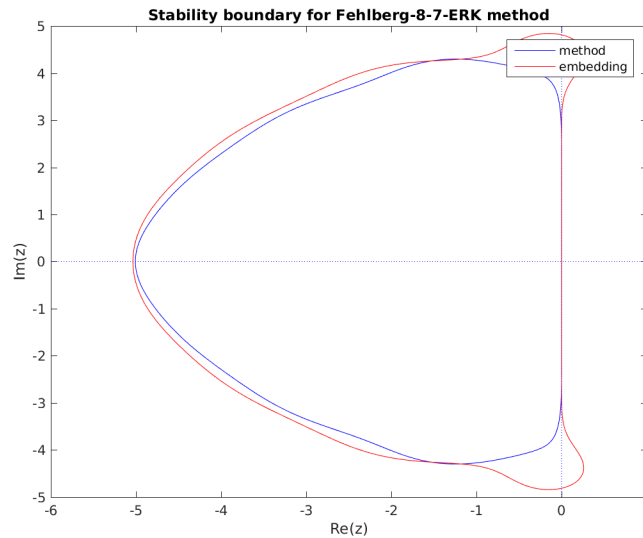


Fig. 19.26: Linear stability region for the Fehlbberg-13-7-8 method. The method's region is outlined in blue; the embedding's region is in red.

`Name()`, `ERKStepSetTableName()` or `ARKodeButcherTable_LoadERKByName()`. This is the default 9th order explicit method (from [121]).

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

## 19.2 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. ARKODE's diagonally-implicit Butcher tables are provided in the enumeration

enum **ARKODE\_DIRKTableID**

with values specified in Table 19.2.

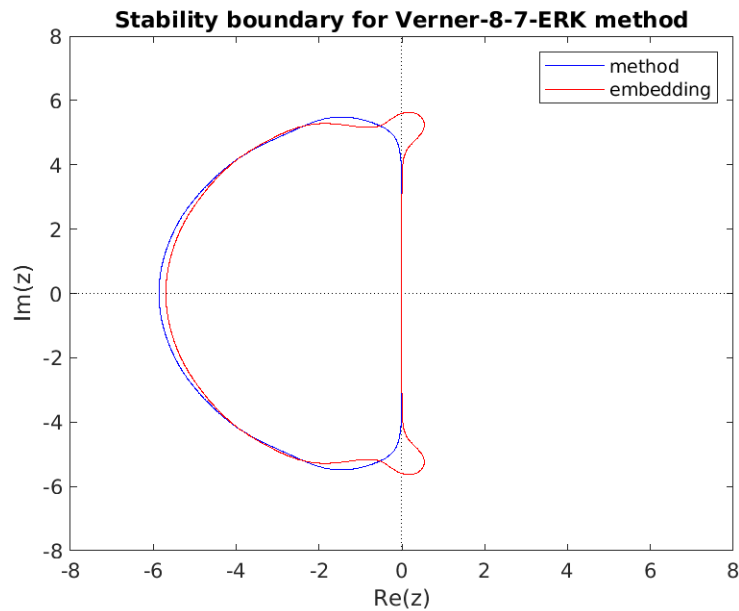


Fig. 19.27: Linear stability region for the Verner-16-8-9 method. The method’s region is outlined in blue; the embedding’s region is in red.

Table 19.2: Implicit Butcher tables. The default method for each order is marked with an asterisk (\*).

Method ID	Stages	Embedded Order	Order
ARKODE_BACKWARD_EULER_1_1	1	—	1*
ARKODE_ARK2_DIRK_3_1_2	3	1	2*
ARKODE_SDIRK_2_1_2	2	1	2
ARKODE_IMPLICIT_MIDPOINT_1_2	1	—	2
ARKODE_IMPLICIT_TRAPEZOIDAL_2_2	2	—	2
ARKODE_BILLINGTON_3_3_2	3	3	2
ARKODE_TRBDF2_3_3_2	3	3	2
ARKODE_ESDIRK325L2SA_5_2_3	5	2	3*
ARKODE_ESDIRK324L2SA_4_2_3	4	2	3
ARKODE_ESDIRK32I5L2SA_5_2_3	5	2	3
ARKODE_KVAERNO_4_2_3	4	2	3
ARKODE_ARK324L2SA_DIRK_4_2_3	4	2	3
ARKODE_ESDIRK436L2SA_6_3_4	6	3	4*
ARKODE_CASH_5_2_4	5	2	4
ARKODE_CASH_5_3_4	5	3	4
ARKODE_SDIRK_5_3_4	5	3	4
ARKODE_KVAERNO_5_3_4	5	3	4
ARKODE_ARK436L2SA_DIRK_6_3_4	6	3	4
ARKODE_ARK437L2SA_DIRK_7_3_4	7	3	4
ARKODE_ESDIRK43I6L2SA_6_3_4	6	3	4
ARKODE_QESDIRK436L2SA_6_3_4	6	3	4
ARKODE_ESDIRK437L2SA_7_3_4	7	3	4
ARKODE_ESDIRK547L2SA2_7_4_5	7	4	5*
ARKODE_KVAERNO_7_4_5	7	4	5
ARKODE_ARK548L2SA_DIRK_8_4_5	8	4	5
ARKODE_ARK548L2SAb_DIRK_8_4_5	8	4	5
ARKODE_ESDIRK547L2SA_7_4_5	7	4	5

enumerator **ARKODE\_BACKWARD\_EULER\_1\_1**

Accessible via the constant `ARKODE_BACKWARD_EULER_1_1` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_BACKWARD\_EULER\_1\_1" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the default 1st order implicit method. The method is A-, L-, and B-stable.

$$\begin{array}{c|c} 1 & 1 \\ \hline 1 & 1 \end{array}$$

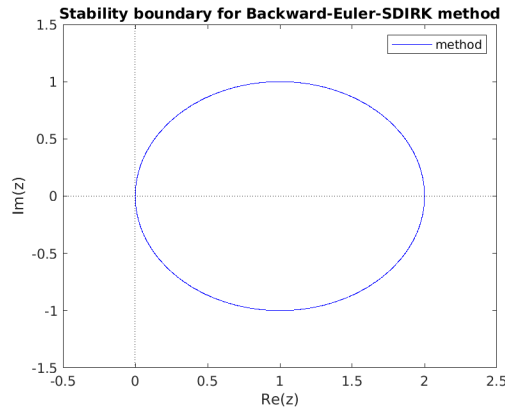


Fig. 19.28: Linear stability region for the backward Euler method.

enumerator **ARKODE\_ARK2\_DIRK\_3\_1\_2**

Accessible via the constant `ARKODE_ARK2_DIRK_3_1_2` to `ARKStepSetTableNum()`, or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ARK2\_DIRK\_3\_1\_2" to `ARKStepSetTableName()`, or `ARKodeButcherTable_LoadDIRKByName()`. This is the default 2nd order implicit method and the implicit portion of the default 2nd order additive method (the implicit portion of the ARK2 method from [50]).

Changed in version 6.3.0: Made the default 2nd order implicit method

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 2 - \sqrt{2} & 1 - \frac{1}{\sqrt{2}} & 1 - \frac{1}{\sqrt{2}} & 0 \\ \hline 1 & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & 1 - \frac{1}{\sqrt{2}} \\ 2 & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & 1 - \frac{1}{\sqrt{2}} \\ 1 & \frac{4-\sqrt{2}}{8} & \frac{4+\sqrt{2}}{8} & \frac{1}{2\sqrt{2}} \end{array}$$

enumerator **ARKODE\_SDIRK\_2\_1\_2**

Accessible via the constant `ARKODE_SDIRK_2_1_2` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_SDIRK\_2\_1\_2" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A- and B-stable.

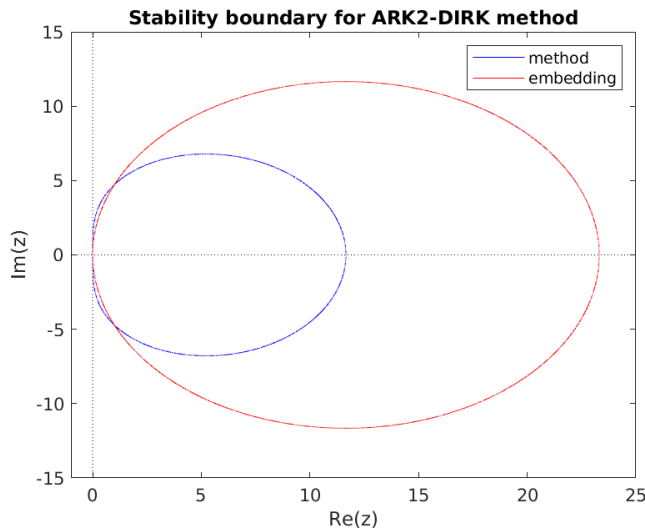


Fig. 19.29: Linear stability region for the ARK2-DIRK method. The method's region is outlined in blue; the embedding's region is in red.

Changed in version 6.3.0: Replaced by `ARKODE_ARK2_DIRK_3_1_2` as the default 2nd order implicit method

$$\begin{array}{c|cc} 1 & 1 & 0 \\ 0 & -1 & 1 \\ \hline 2 & \frac{1}{2} & \frac{1}{2} \\ 1 & 1 & 0 \end{array}$$

enumerator `ARKODE_IMPLICIT_MIDPOINT_1_2`

Accessible via the constant `ARKODE_IMPLICIT_MIDPOINT_1_2` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_IMPLICIT\_MIDPOINT\_1\_2" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. The method is A- and B-stable.

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline 2 & 1 \end{array}$$

enumerator `ARKODE_IMPLICIT_TRAPEZOIDAL_2_2`

Accessible via the constant `ARKODE_IMPLICIT_TRAPEZOIDAL_2_2` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_IMPLICIT\_TRAPEZOIDAL\_2\_2" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. The method is A-stable.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline 2 & \frac{1}{2} & \frac{1}{2} \end{array}$$



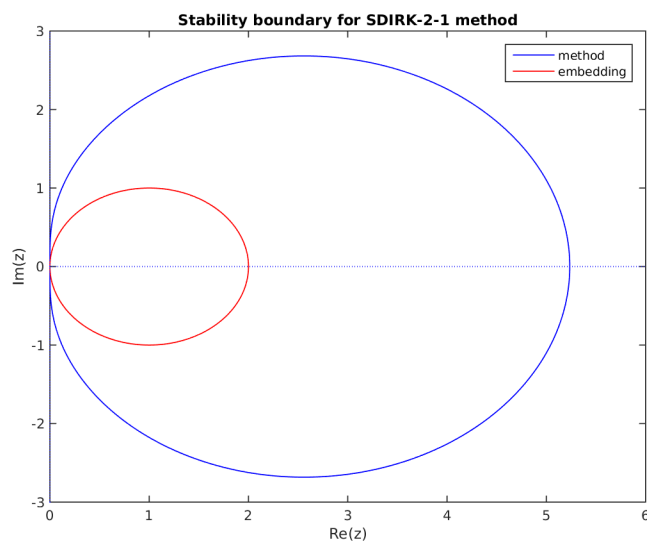


Fig. 19.30: Linear stability region for the SDIRK-2-1-2 method. The method's region is outlined in blue; the embedding's region is in red.

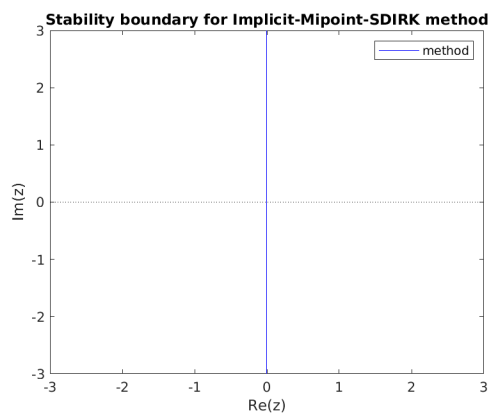


Fig. 19.31: Linear stability region for the implicit midpoint method.

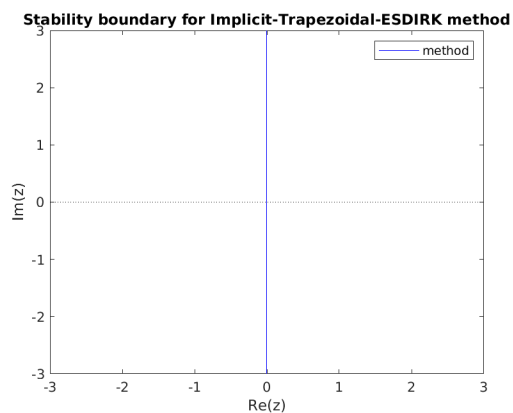


Fig. 19.32: Linear stability region for the implicit trapezoidal method.

enumerator **ARKODE\_BILLINGTON\_3\_3\_2**

Accessible via the constant `ARKODE_BILLINGTON_3_3_2` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_BILLINGTON\_3\_3\_2" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Here, the higher-order embedding is less stable than the lower-order method (from [17]).

$1 - \frac{1}{\sqrt{2}}$	$1 - \frac{1}{\sqrt{2}}$	0	0
$\frac{27}{\sqrt{2}} - 18$	$14\sqrt{2} - 19$	$1 - \frac{1}{\sqrt{2}}$	0
$2 - \frac{1}{\sqrt{2}}$	$\frac{53-5\sqrt{2}}{62}$	$\frac{9+5\sqrt{2}}{62}$	$1 - \frac{1}{\sqrt{2}}$
2	$\frac{53-5\sqrt{2}}{62}$	$\frac{9+5\sqrt{2}}{62}$	0
3	$\frac{263-95\sqrt{2}}{186}$	$\frac{47+33\sqrt{2}}{186}$	$\frac{\sqrt{2}-2}{3}$

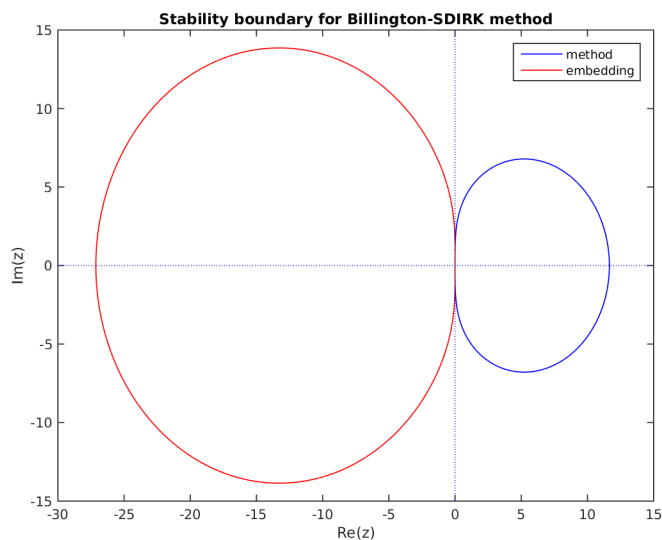


Fig. 19.33: Linear stability region for the Billington method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_TRBDF2\_3\_3\_2**

Accessible via the constant `ARKODE_TRBDF2_3_3_2` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_TRBDF2\_3\_3\_2" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. As with Billington, here the higher-order embedding is less stable than the lower-order method (from [16]).

0	0	0	0
$2 - \sqrt{2}$	$\frac{2-\sqrt{2}}{2}$	$\frac{2-\sqrt{2}}{2}$	0
1	$\frac{\sqrt{2}}{4}$	$\frac{\sqrt{2}}{4}$	$\frac{2-\sqrt{2}}{2}$
2	$\frac{\sqrt{2}}{4}$	$\frac{\sqrt{2}}{4}$	$\frac{2-\sqrt{2}}{2}$
3	$\frac{1-\sqrt{2}}{3}$	$\frac{3\sqrt{2}+1}{3}$	$\frac{2-\sqrt{2}}{6}$

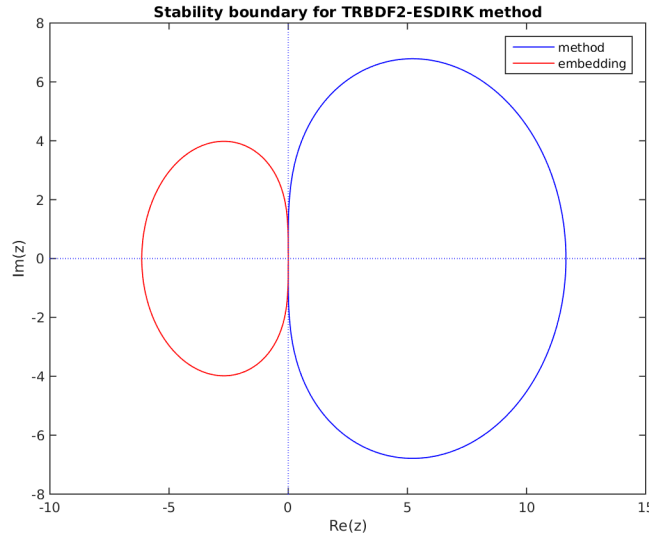


Fig. 19.34: Linear stability region for the TRBDF2 method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ESDIRK325L2SA\_5\_2\_3**

Accessible via the constant `ARKODE_ESDIRK325L2SA_5_2_3` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK325L2SA\_5\_2\_3" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the default 3rd order implicit method and the ESDIRK3(2)5L[2]SA method from [69]. Both the method and embedding are A- and L-stable.

Changed in version 6.3.0: Made the default 3rd order implicit method

enumerator **ARKODE\_ESDIRK324L2SA\_4\_2\_3**

Accessible via the constant `ARKODE_ESDIRK324L2SA_4_2_3` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK324L2SA\_4\_2\_3" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the ESDIRK3(2)4L[2]SA method from [70]. Both the method and embedding are A- and L-stable.

enumerator **ARKODE\_ESDIRK32I5L2SA\_5\_2\_3**

Accessible via the constant `ARKODE_ESDIRK32I5L2SA_5_2_3` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK32I5L2SA\_5\_2\_3" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the ESDIRK3(2I)5L[2]SA method from [69]. Both the method and embedding are A- and L-stable.

enumerator **ARKODE\_KVAERNO\_4\_2\_3**

Accessible via the constant `ARKODE_KVAERNO_4_2_3` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_KVAERNO\_4\_2\_3" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A-stable; additionally the method is L-

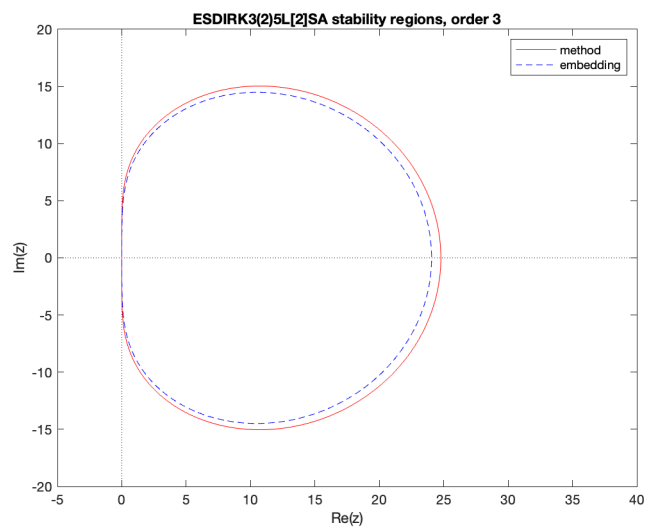


Fig. 19.35: Linear stability region for the ESDIRK325L2SA-5-2-3 method method. The method's region is outlined in blue; the embedding's region is in red.

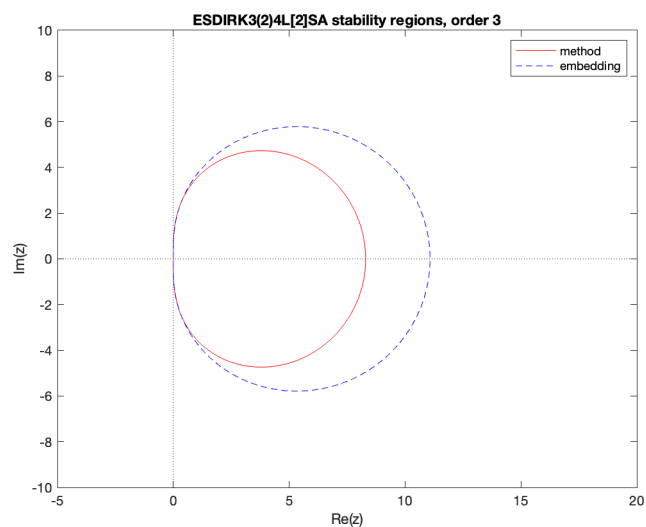


Fig. 19.36: Linear stability region for the ESDIRK324L2SA-4-2-3 method method. The method's region is outlined in blue; the embedding's region is in red.

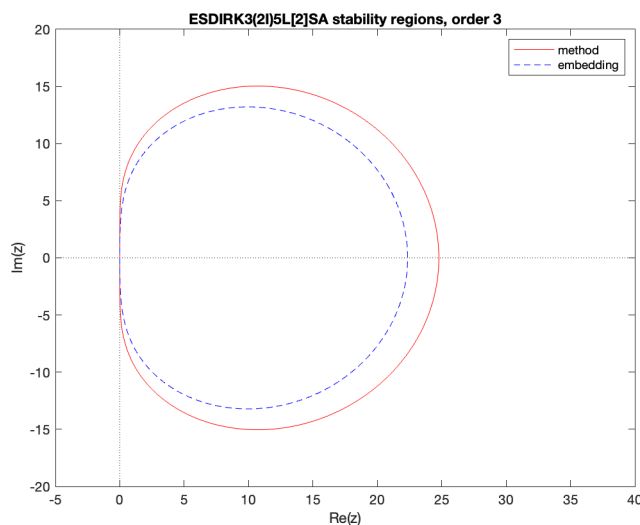


Fig. 19.37: Linear stability region for the ESDIRK3(2)5L2SA-5-2-3 method method. The method's region is outlined in blue; the embedding's region is in red.

stable (from [75]).

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

enumerator **ARKODE\_ARK324L2SA\_DIRK\_4\_2\_3**

Accessible via the constant **ARKODE\_ARK324L2SA\_DIRK\_4\_2\_3** to *ARKStepSetTableNum()* or *ARKodeButcherTable\_LoadDIRK()*. Accessible via the string "ARKODE\_ARK324L2SA\_DIRK\_4\_2\_3" to *ARKStepSetTableName()* or *ARKodeButcherTable\_LoadDIRKByName()*. This is the implicit portion of the default 3rd order additive method. Both the method and embedding are A-stable; additionally the method is L-stable (this is the implicit portion of the ARK3(2)4L[2]SA method from [68]).

Changed in version 6.3.0: Replaced by **ARKODE\_ESDIRK325L2SA\_5\_2\_3** as the default 3rd order implicit method

0	0	0	0	0
<u>1767732205903</u> 2027836641118	<u>1767732205903</u> 4055673282236	<u>1767732205903</u> 4055673282236	0	0
<u>3</u> 5	<u>2746238789719</u> 10658868560708	<u>640167445237</u> 6845629431997	<u>1767732205903</u> 4055673282236	0
1	<u>1471266399579</u> 7840856788654	<u>4482444167858</u> 7529755066697	<u>11266239266428</u> 11593286722821	<u>1767732205903</u> 4055673282236
3	<u>1471266399579</u> 7840856788654	<u>4482444167858</u> 7529755066697	<u>11266239266428</u> 11593286722821	<u>1767732205903</u> 4055673282236
2	<u>2756255671327</u> 12835298489170	<u>10771552573575</u> 22201958757719	<u>9247589265047</u> 10645013368117	<u>2193209047091</u> 5459859503100

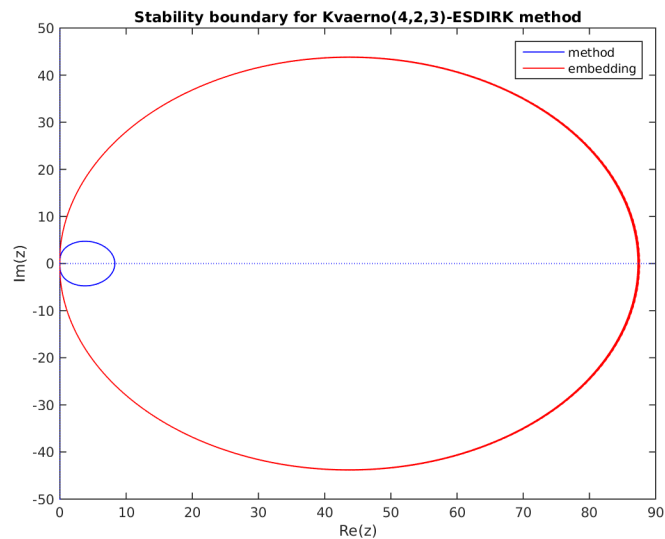


Fig. 19.38: Linear stability region for the Kvaerno-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

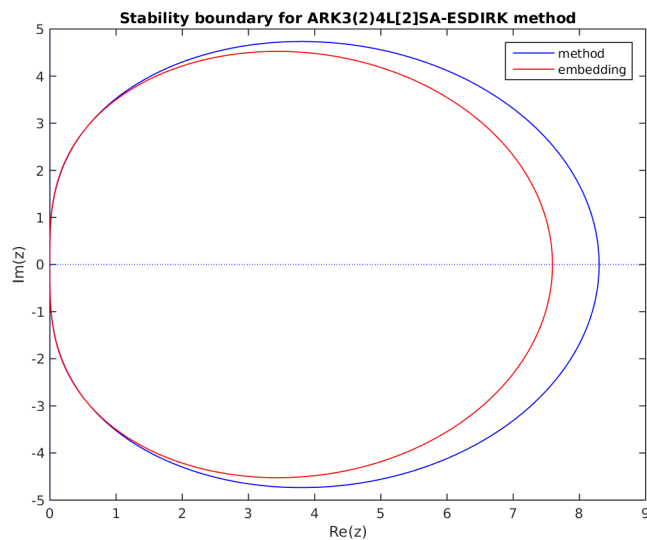


Fig. 19.39: Linear stability region for the implicit ARK324L2SA-DIRK-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ESDIRK436L2SA\_6\_3\_4**

Accessible via the constant `ARKODE_ESDIRK436L2SA_6_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK436L2SA\_6\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the default 4th order implicit method and the ESDIRK4(3)6L[2]SA method from [69]. Both the method and embedding are A- and L-stable.

Changed in version 6.3.0: Made the default 4th order implicit method

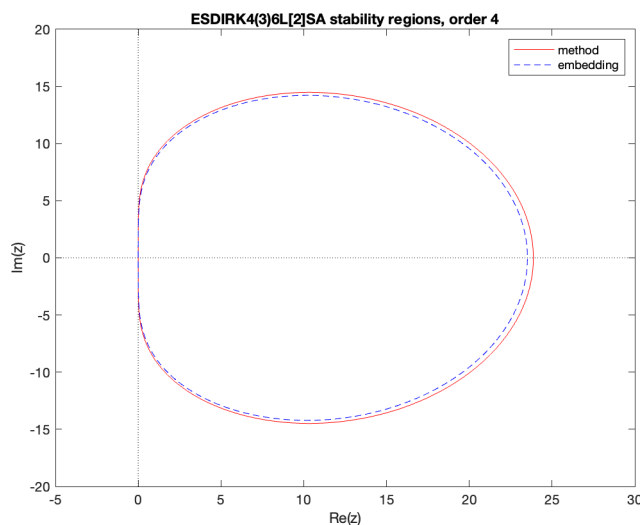


Fig. 19.40: Linear stability region for the ESDIRK436L2SA-6-3-4 method method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_CASH\_5\_2\_4**

Accessible via the constant `ARKODE_CASH_5_2_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_CASH\_5\_2\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A-stable; additionally the method is L-stable (from [28]).

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

enumerator **ARKODE\_CASH\_5\_3\_4**

Accessible via the constant `ARKODE_CASH_5_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_CASH\_5\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A-stable; additionally the method is L-

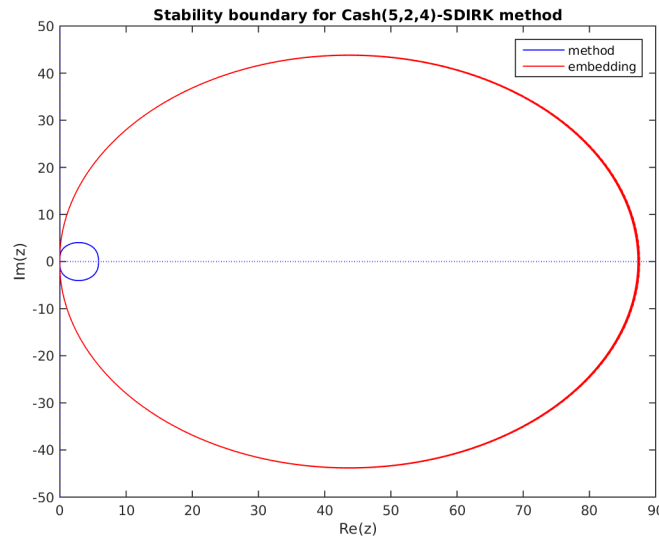


Fig. 19.41: Linear stability region for the Cash-5-2-4 method. The method's region is outlined in blue; the embedding's region is in red.

stable (from [28]).

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

enumerator **ARKODE\_SDIRK\_5\_3\_4**

Accessible via the constant **ARKODE\_SDIRK\_5\_3\_4** to [ARKStepSetTableNum\(\)](#) or [ARKodeButcherTable\\_LoadDIRK\(\)](#). Accessible via the string "ARKODE\_SDIRK\_5\_3\_4" to [ARKStepSetTableName\(\)](#) or [ARKodeButcherTable\\_LoadDIRKByName\(\)](#). Here, the method is both A- and L-stable, although the embedding has reduced stability (from [56]).

Changed in version 6.3.0: Replaced by **ARKODE\_ESDIRK436L2SA\_6\_3\_4** as the default 4th order implicit method

$\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0
$\frac{3}{4}$	$\frac{1}{2}$	$\frac{1}{4}$	0	0	0
$\frac{11}{20}$	$\frac{17}{50}$	$-\frac{1}{25}$	$\frac{1}{4}$	0	0
$\frac{1}{2}$	$\frac{371}{1360}$	$-\frac{137}{2720}$	$\frac{15}{544}$	$\frac{1}{4}$	0
1	$\frac{25}{24}$	$-\frac{49}{48}$	$\frac{125}{16}$	$-\frac{85}{12}$	$\frac{1}{4}$
4	$\frac{25}{24}$	$-\frac{49}{48}$	$\frac{125}{16}$	$-\frac{85}{12}$	$\frac{1}{4}$
3	$\frac{59}{48}$	$-\frac{17}{96}$	$\frac{225}{32}$	$-\frac{85}{12}$	0



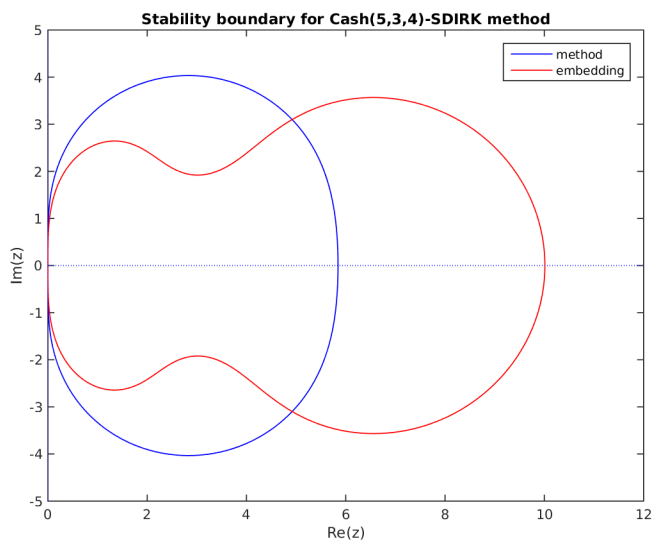


Fig. 19.42: Linear stability region for the Cash-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_KVAERNO\_5\_3\_4**

Accessible via the constant `ARKODE_KVAERNO_5_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_KVAERNO\_5\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A-stable (from [75]).

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator **ARKODE\_ARK436L2SA\_DIRK\_6\_3\_4**

Accessible via the constant `ARKODE_ARK436L2SA_DIRK_6_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ARK436L2SA\_DIRK\_6\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A-stable; additionally the method is L-stable (this is the implicit portion of the ARK4(3)6L[2]SA method from [68]).

Changed in version 6.3.0: Replaced by `ARKODE_ARK437L2SA_DIRK_7_3_4` as the implicit portion of the default 4th order additive method

0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0
$\frac{83}{250}$	$\frac{8611}{62500}$	$-\frac{1743}{31250}$	$\frac{1}{4}$	0	0	0
$\frac{31}{50}$	$\frac{5012029}{34652500}$	$-\frac{654441}{2922500}$	$\frac{174375}{388108}$	$\frac{1}{4}$	0	0
$\frac{17}{20}$	$\frac{15267082809}{155376265600}$	$-\frac{71443401}{120774400}$	$\frac{730878875}{902184768}$	$\frac{2285395}{8070912}$	$\frac{1}{4}$	0
1	$\frac{82889}{524892}$	0	$\frac{15625}{83664}$	$\frac{69875}{102672}$	$-\frac{2260}{8211}$	$\frac{1}{4}$
4	$\frac{82889}{524892}$	0	$\frac{15625}{83664}$	$\frac{69875}{102672}$	$-\frac{2260}{8211}$	$\frac{1}{4}$
3	$\frac{4586570599}{29645900160}$	0	$\frac{178811875}{945068544}$	$\frac{814220225}{1159782912}$	$-\frac{3700637}{11593932}$	$\frac{61727}{225920}$

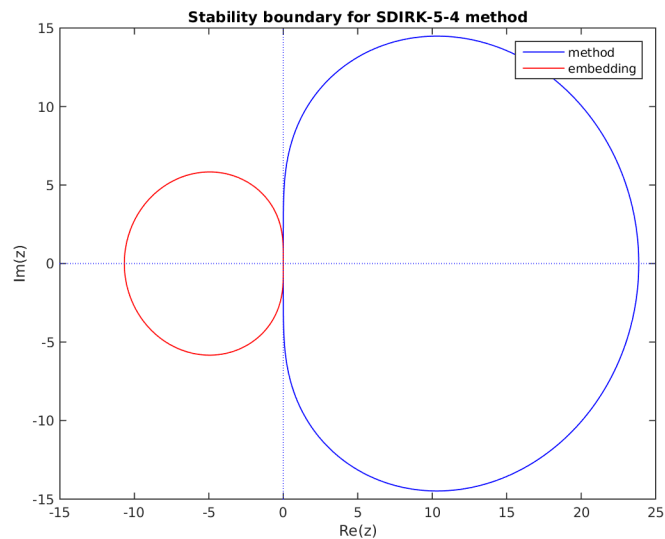


Fig. 19.43: Linear stability region for the SDIRK-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

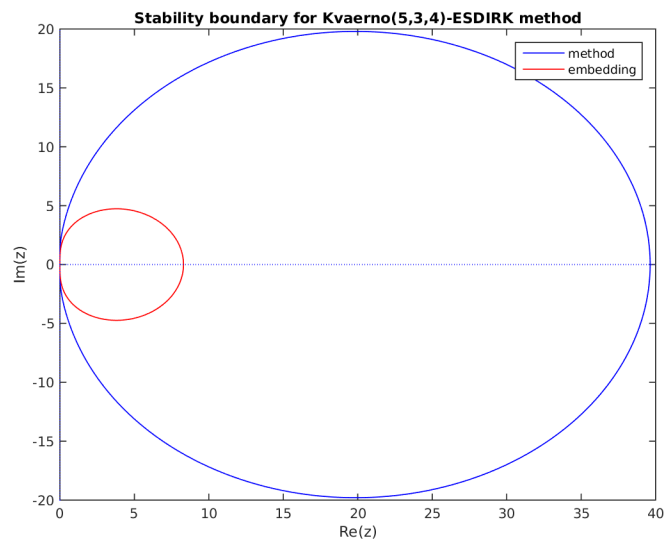


Fig. 19.44: Linear stability region for the Kvaerno-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

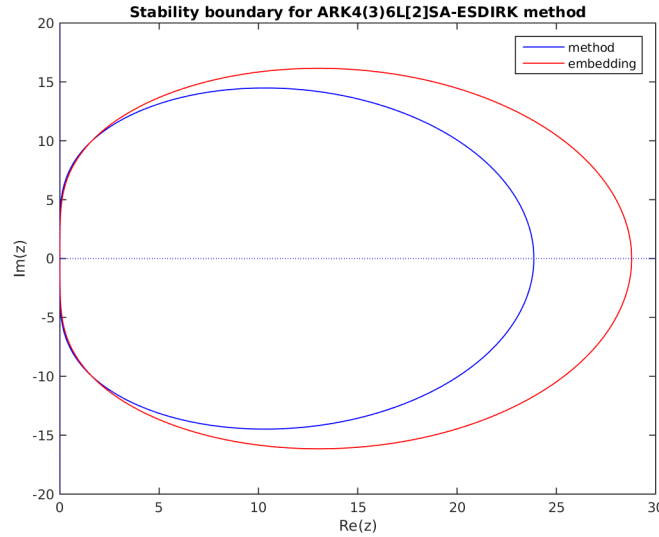


Fig. 19.45: Linear stability region for the ARK436L2SA-Dirk-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ARK437L2SA\_DIRK\_7\_3\_4**

Accessible via the constant `ARKODE_ARK437L2SA_DIRK_7_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ARK437L2SA\_DIRK\_7\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the implicit portion of the default 4th order additive method and the implicit portion of the 4th order ARK4(3)7L[2]SA method from [71]. Both the method and embedding are A- and L-stable.

Changed in version 6.3.0: Made the implicit portion of the default 4th order additive method

0	0	0	0	0	0	0	0
$\frac{247}{1000}$	$\frac{1235}{10000}$	$\frac{1235}{10000}$	0	0	0	0	0
$\frac{4276536705230}{10142255878289}$	$\frac{624185399699}{4186980696204}$	$\frac{624185399699}{4186980696204}$	$\frac{1235}{10000}$	0	0	0	0
$\frac{67}{200}$	$\frac{1258591069120}{10082082980243}$	$\frac{1258591069120}{10082082980243}$	$-\frac{322722984531}{8455138723562}$	$\frac{1235}{10000}$	0	0	0
$\frac{3}{40}$	$-\frac{436103496990}{5971407786587}$	$-\frac{436103496990}{5971407786587}$	$-\frac{2689175662187}{11046760208243}$	$\frac{4431412449334}{12995360898505}$	$\frac{1235}{10000}$	0	0
$\frac{7}{10}$	$-\frac{2207373168298}{14430576638973}$	$-\frac{2207373168298}{14430576638973}$	$\frac{242511121179}{3358618340039}$	$\frac{3145666661981}{7780404714551}$	$\frac{5882073923981}{14490790706663}$	$\frac{1235}{10000}$	0
1	0	0	$\frac{9164257142617}{17756377923965}$	$-\frac{10812980402763}{74029279521829}$	$\frac{1335994250573}{5691609445217}$	$\frac{2273837961795}{8368240463276}$	$\frac{1235}{10000}$
4	0	0	$\frac{9164257142617}{17756377923965}$	$-\frac{10812980402763}{74029279521829}$	$\frac{1335994250573}{5691609445217}$	$\frac{2273837961795}{8368240463276}$	$\frac{1235}{10000}$
3	0	0	$\frac{4469248916618}{8635866897933}$	$-\frac{621260224600}{4094290005349}$	$\frac{696572312987}{2942599194819}$	$\frac{1532940081127}{5565293938103}$	$\frac{2441}{20000}$

enumerator **ARKODE\_ESDIRK43I6L2SA\_6\_3\_4**

Accessible via the constant `ARKODE_ESDIRK43I6L2SA_6_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK43I6L2SA\_6\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the ESDIRK4(3I)6L[2]SA method from [69]. Both the method and embedding are A- and L-stable.

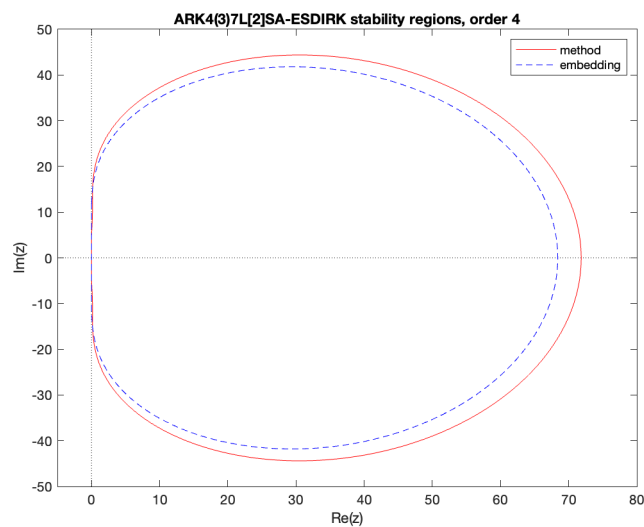


Fig. 19.46: Linear stability region for the ARK437L2SA-DIRK-7-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

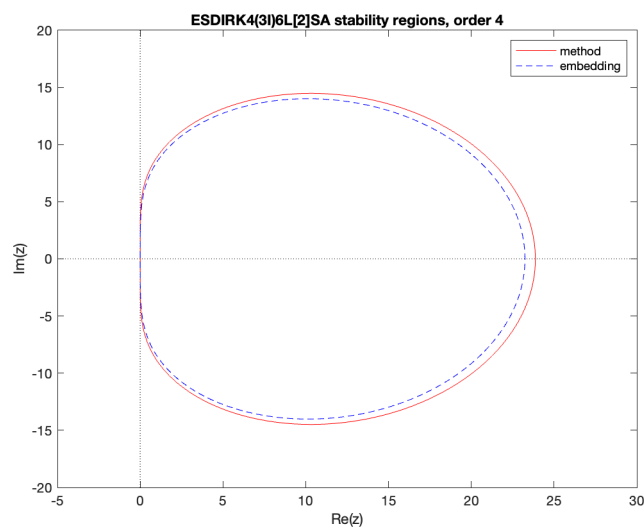


Fig. 19.47: Linear stability region for the ESDIRK43I6L2SA-6-3-4 method method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_QESDIRK436L2SA\_6\_3\_4**

Accessible via the constant `ARKODE_QESDIRK436L2SA_6_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_QESDIRK436L2SA\_6\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the QESDIRK4(3)6L[2]SA method from [69]. Both the method and embedding are A- and L-stable.

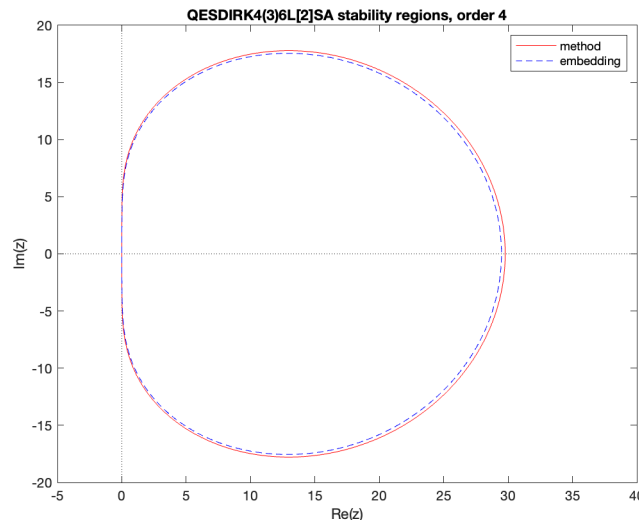


Fig. 19.48: Linear stability region for the QESDIRK436L2SA-6-3-4 method method. The method's region is outlined in blue; the embedding's region is in red.

enumerator **ARKODE\_ESDIRK437L2SA\_7\_3\_4**

Accessible via the constant `ARKODE_ESDIRK437L2SA_7_3_4` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK437L2SA\_7\_3\_4" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the ESDIRK4(3)7L[2]SA method from [70]. Both the method and embedding are A- and L-stable.

enumerator **ARKODE\_ESDIRK547L2SA2\_7\_4\_5**

Accessible via the constant `ARKODE_ESDIRK547L2SA2_7_4_5` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK547L2SA2\_7\_4\_5" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the default 5th order implicit method and the ESDIRK5(4)7L[2]SA2 method from [70]. Both the method and embedding are A- and L-stable.

Changed in version 6.3.0: Made the default 5th order implicit method

enumerator **ARKODE\_KVAERNO\_7\_4\_5**

Accessible via the constant `ARKODE_KVAERNO_7_4_5` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_KVAERNO\_7\_4\_5" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A-stable; additionally the method is L-stable (from [75]).

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator **ARKODE\_ARK548L2SA\_DIRK\_8\_4\_5**

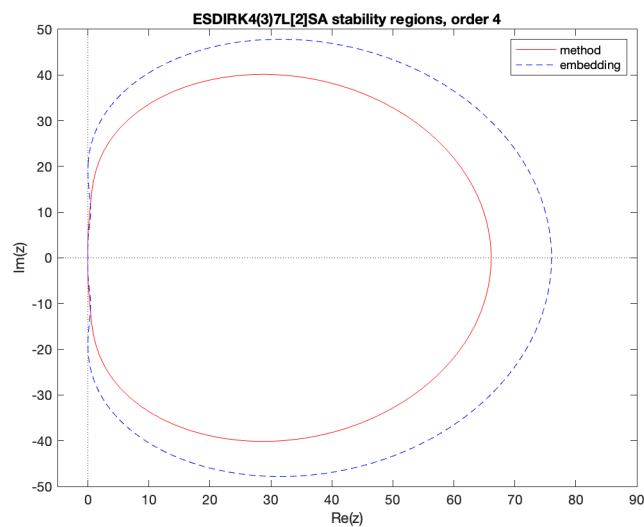


Fig. 19.49: Linear stability region for the ESDIRK437L2SA-7-3-4 method method. The method's region is outlined in blue; the embedding's region is in red.

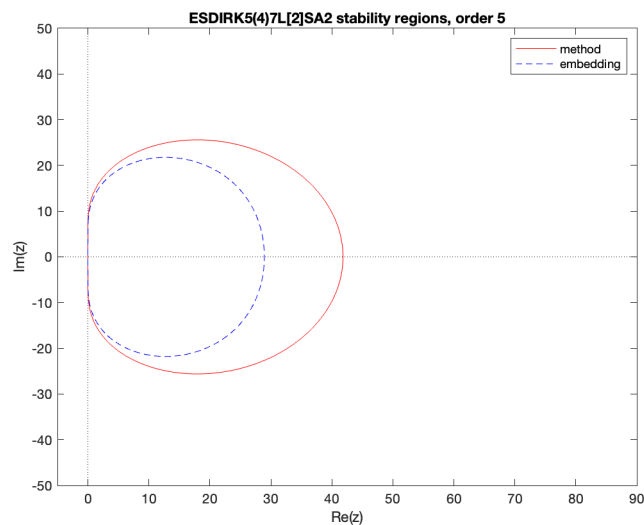


Fig. 19.50: Linear stability region for the ESDIRK547L2SA2-7-4-5 method method. The method's region is outlined in blue; the embedding's region is in red.

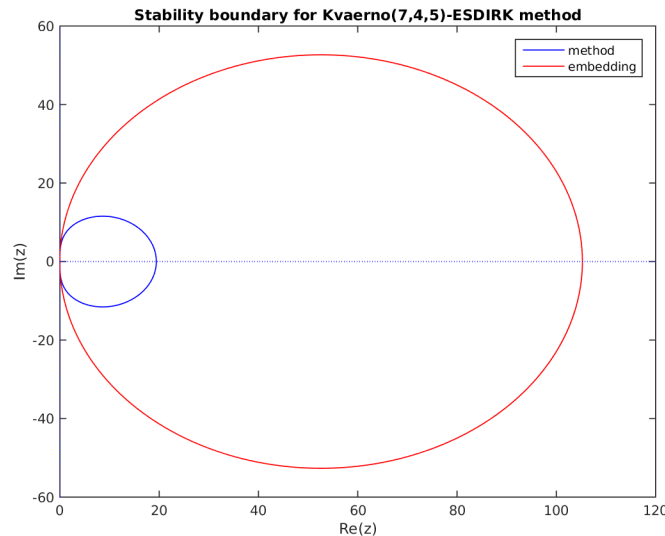


Fig. 19.51: Linear stability region for the Kvaerno-7-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

Accessible via the constant `ARKODE_ARK548L2SA_DIRK_8_4_5` for `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ARK548L2SA\_DIRK\_8\_4\_5" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. Both the method and embedding are A-stable; additionally the method is L-stable (the implicit portion of the ARK5(4)8L[2]SA method from [68]).

Changed in version 6.3.0: Replaced by `ARKODE_ESDIRK547L2SA2_7_4_5` as the default 5th order implicit method and replaced by `ARKODE_ARK548L2SAb_DIRK_8_4_5` as the implicit portion of the default 5th order additive method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator **ARKODE\_ARK548L2SAb\_DIRK\_8\_4\_5**

Accessible via the constant `ARKODE_ARK548L2SAb_DIRK_8_4_5` for `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ARK548L2SAb\_DIRK\_8\_4\_5" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the implicit portion of the default 5th order additive method. Both the method and embedding are A-stable; additionally the method is L-stable (this is the implicit portion of the 5th order ARK5(4)8L[2]SA method from [71]).

Changed in version 6.3.0: Made the implicit portion of the default 5th order additive method

The Butcher table is too large to fit in the PDF version of this documentation. Please see the HTML documentation for the table coefficients.

enumerator **ARKODE\_ESDIRK547L2SA\_7\_4\_5**

Accessible via the constant `ARKODE_ESDIRK547L2SA_7_4_5` to `ARKStepSetTableNum()` or `ARKodeButcherTable_LoadDIRK()`. Accessible via the string "ARKODE\_ESDIRK547L2SA\_7\_4\_5" to `ARKStepSetTableName()` or `ARKodeButcherTable_LoadDIRKByName()`. This is the ESDIRK5(4)7L[2]SA method from [69]. Both the method and embedding are A- and L-stable.

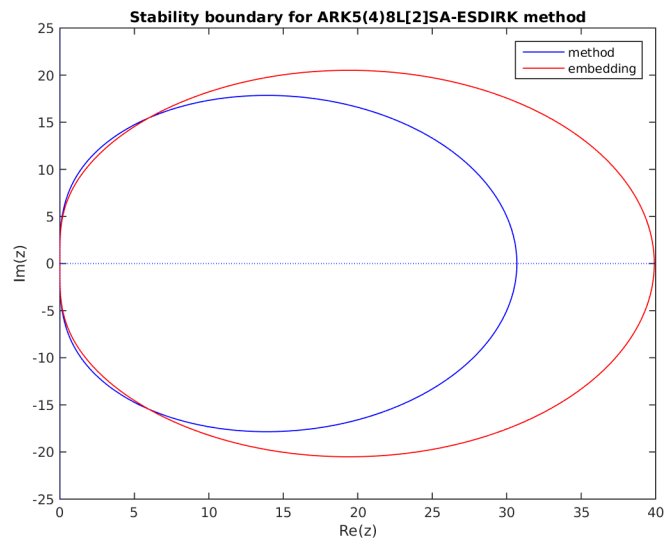


Fig. 19.52: Linear stability region for the implicit ARK548L2SA-ESDIRK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

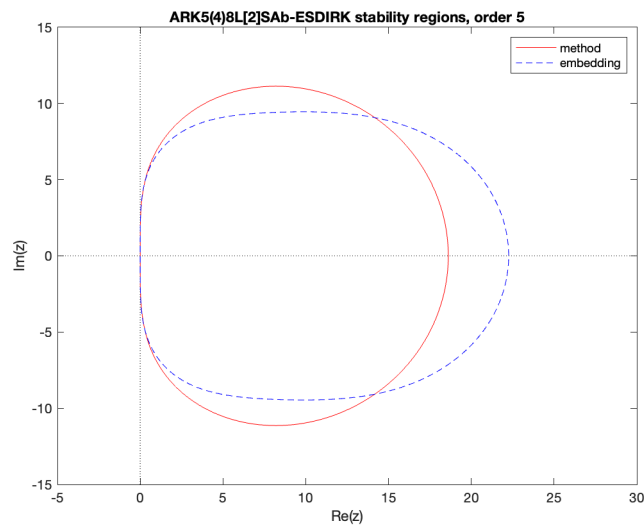


Fig. 19.53: Linear stability region for the ARK548L2SAb-ESDIRK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.



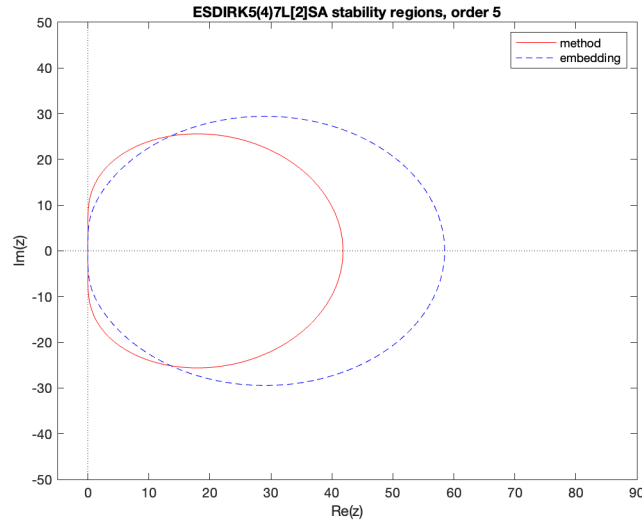


Fig. 19.54: Linear stability region for the ESDIRK547L2SA-7-4-5 method method. The method's region is outlined in blue; the embedding's region is in red.

## 19.3 Additive Butcher tables

In the category of additive Runge–Kutta methods for split implicit and explicit calculations, ARKODE includes methods that have orders 2 through 5, with embeddings that are of orders 1 through 4. These Butcher table pairs are as follows:

Table 19.3: Additive Butcher tables. The default method for each order is marked with an asterisk (\*).

ERK Method ID	DIRK Method ID	Stages	Embedded Order	Order
<a href="#">ARKODE_ARK2_ERK_3_1_2</a>	<a href="#">ARKODE_ARK2_DIRK_3_1_2</a>	3	1	2*
<a href="#">ARKODE_ARK324L2SA_ERK_4_2_3</a>	<a href="#">ARKODE_ARK324L2SA_DIRK_4_2_3</a>	4	2	3*
<a href="#">ARKODE_ARK437L2SA_ERK_7_3_4</a>	<a href="#">ARKODE_ARK437L2SA_DIRK_7_3_4</a>	7	3	4*
<a href="#">ARKODE_ARK436L2SA_ERK_6_3_4</a>	<a href="#">ARKODE_ARK436L2SA_DIRK_6_3_4</a>	6	3	4
<a href="#">ARKODE_ARK548L2SAb_ERK_8_4_5</a>	<a href="#">ARKODE_ARK548L2SAb_DIRK_8_4_5</a>	8	4	5*
<a href="#">ARKODE_ARK548L2SA_ERK_8_4_5</a>	<a href="#">ARKODE_ARK548L2SA_DIRK_8_4_5</a>	8	5	5

## 19.4 Symplectic Partitioned Butcher tables

In the category of symplectic partitioned Runge–Kutta (SPRK) methods, ARKODE includes methods that have orders  $q = \{1, 2, 3, 4, 5, 6, 8, 10\}$ . ARKODE's symplectic partitioned Butcher tables are provided in the enumeration

enum **ARKODE\_SPRKMethodID**

with values specified in Table 19.4.

Table 19.4: Symplectic partitioned Butcher tables. The default method for each order is marked with an asterisk (\*).

Method ID	Stages	Order
<a href="#"><i>ARKODE_SPRK_EULER_1_1</i></a>	1	1*
<a href="#"><i>ARKODE_SPRK_LEAPFROG_2_2</i></a>	2	2*
<a href="#"><i>ARKODE_SPRK_PSEUDO_LEAPFROG_2_2</i></a>	2	2
<a href="#"><i>ARKODE_SPRK_MCLACHLAN_2_2</i></a>	2	2
<a href="#"><i>ARKODE_SPRK_MCLACHLAN_3_3</i></a>	3	3*
<a href="#"><i>ARKODE_SPRK_RUTH_3_3</i></a>	3	3
<a href="#"><i>ARKODE_SPRK_MCLACHLAN_4_4</i></a>	4	4*
<a href="#"><i>ARKODE_SPRK_CANDY_ROZMUS_4_4</i></a>	4	4
<a href="#"><i>ARKODE_SPRK_MCLACHLAN_5_6</i></a>	6	5*
<a href="#"><i>ARKODE_SPRK_YOSHIDA_6_8</i></a>	8	6*
<a href="#"><i>ARKODE_SPRK_SUZUKI_UMENO_8_16</i></a>	16	8*
<a href="#"><i>ARKODE_SPRK_SOFRONIOU_10_36</i></a>	36	10*

enumerator **ARKODE\_SPRK\_EULER\_1\_1**

Accessible via the constant (or string) `ARKODE_SPRK_EULER_1_1` to [\*ARKodeSPRKTable\\_Load\(\)\*](#) or [\*ARKodeSPRKTable\\_LoadByName\(\)\*](#). This is the classic Symplectic Euler method and the default 1st order method.

enumerator **ARKODE\_SPRK\_LEAPFROG\_2\_2**

Accessible via the constant (or string) `ARKODE_SPRK_LEAPFROG_2_2` to [\*ARKodeSPRKTable\\_Load\(\)\*](#) or [\*ARKodeSPRKTable\\_LoadByName\(\)\*](#). This is the classic Leapfrog/Verlet method and the default 2nd order method.

enumerator **ARKODE\_SPRK\_PSEUDO\_LEAPFROG\_2\_2**

Accessible via the constant (or string) `ARKODE_SPRK_PSEUDO_LEAPFROG_2_2` to [\*ARKodeSPRKTable\\_Load\(\)\*](#) or [\*ARKodeSPRKTable\\_LoadByName\(\)\*](#). This is the classic Pseudo Leapfrog/Verlet method.

enumerator **ARKODE\_SPRK\_MCLACHLAN\_2\_2**

Accessible via the constant (or string) `ARKODE_SPRK_MCLACHLAN_2_2` to [\*ARKodeSPRKTable\\_Load\(\)\*](#) or [\*ARKodeSPRKTable\\_LoadByName\(\)\*](#). This is the 2nd order method given by McLachlan in [81].

enumerator **ARKODE\_SPRK\_MCLACHLAN\_3\_3**

Accessible via the constant (or string) `ARKODE_SPRK_MCLACHLAN_3_3` to [\*ARKodeSPRKTable\\_Load\(\)\*](#) or [\*ARKodeSPRKTable\\_LoadByName\(\)\*](#). This is the 3rd order method given by McLachlan in [81] and the default 3rd order method.

enumerator **ARKODE\_SPRK\_RUTH\_3\_3**

Accessible via the constant (or string) `ARKODE_SPRK_RUTH_3_3` to [\*ARKodeSPRKTable\\_Load\(\)\*](#) or [\*ARKodeSPRKTable\\_LoadByName\(\)\*](#). This is the 3rd order method given by Ruth in [90].

enumerator **ARKODE\_SPRK\_MCLACHLAN\_4\_4**

Accessible via the constant (or string) `ARKODE_SPRK_MCLACHLAN_4_4` to [\*ARKodeSPRKTable\\_Load\(\)\*](#) or [\*ARKodeSPRKTable\\_LoadByName\(\)\*](#). This is the 4th order method given by McLachlan in [81] and the default 4th order method.

**Warning**

This method only has coefficients sufficient for single or double precision.

enumerator **ARKODE\_SPRK\_CANDY\_ROZMUS\_4\_4**

Accessible via the constant (or string) `ARKODE_SPRK_CANDY_ROZMUS_4_4` to [ARKodeSPRKTable\\_Load\(\)](#) or [ARKodeSPRKTable\\_LoadByName\(\)](#). This is the 4th order method given by Candy and Rozmus in [27].

enumerator **ARKODE\_SPRK\_MCLACHLAN\_5\_6**

Accessible via the constant (or string) `ARKODE_SPRK_MCLACHLAN_5_6` to [ARKodeSPRKTable\\_Load\(\)](#) or [ARKodeSPRKTable\\_LoadByName\(\)](#). This is the 5th order method given by McLachlan in [81] and the default 5th order method.

**Warning**

This method only has coefficients sufficient for single or double precision.

enumerator **ARKODE\_SPRK\_YOSHIDA\_6\_8**

Accessible via the constant (or string) `ARKODE_SPRK_YOSHIDA_6_8` to [ARKodeSPRKTable\\_Load\(\)](#) or [ARKodeSPRKTable\\_LoadByName\(\)](#). This is the 6th order method given by Yoshida in [126] and the default 6th order method.

enumerator **ARKODE\_SPRK\_SUZUKI\_UMENO\_8\_16**

Accessible via the constant (or string) `ARKODE_SPRK_SUZUKI_UMENO_8_16` to [ARKodeSPRKTable\\_Load\(\)](#) or [ARKodeSPRKTable\\_LoadByName\(\)](#). This is the 8th order method given by Suzuki and Umeno in [113] and the default 8th order method.

enumerator **ARKODE\_SPRK\_SOFRONIOU\_10\_36**

Accessible via the constant (or string) `ARKODE_SPRK_SOFRONIOU_10_36` to [ARKodeSPRKTable\\_Load\(\)](#) or [ARKodeSPRKTable\\_LoadByName\(\)](#). This is the 10th order method given by Sofroniou and Spaletta in [108] and the default 10th order method.



# Chapter 20

## Fortran

SUNDIALS provides modern, Fortran 2003 based, interfaces as Fortran modules to most of the C API (see [Table 20.1](#)).

### Note

Fortran users should first read the *General User Guide*. The Fortran interfaces closely follow the C/C++ usage of SUNDIALS, so the Fortran User Guide primarily covers differences.

## 20.1 Introduction

An interface module can be accessed with the `use` statement, e.g.

```
use fsundials_core_mod      ! this is needed to access core SUNDIALS types, utilities, and data structures
use fcvode_mod              ! this is needed to access CVODE functions and types
use fnvector_openmp_mod     ! this is needed to access the OpenMP implementation of the N_Vector class
```

and by linking to the Fortran 2003 library in addition to the C library, e.g. `libsundials_fcore_mod.<so|a>`, `libsundials_core.<so|a>`, `libsundials_fnvecopenmp_mod.<so|a>`, `libsundials_nvecopenmp.<so|a>`, `libsundials_fcvode_mod.<so|a>` and `libsundials_cvode.<so|a>`. The use statements mirror the `#include` statements needed when using the C API.

The Fortran 2003 interfaces leverage the `iso_c_binding` module and the `bind(C)` attribute to closely follow the SUNDIALS C API (modulo language differences). The SUNDIALS classes, e.g. `N_Vector`, are interfaced as Fortran derived types, and function signatures are matched but with an `F` prepending the name, e.g. `FN_VConst` instead of `N_VConst()` or `FCvodeCreate` instead of `CVodeCreate`. Constants are named exactly as they are in the C API. Accordingly, using SUNDIALS via the Fortran 2003 interfaces looks just like using it in C. Some caveats stemming from the language differences are discussed in [§20.3](#). A discussion on the topic of equivalent data types in C and Fortran 2003 is presented in [§20.2](#).

Further information on the Fortran 2003 interfaces specific to the `N_Vector`, `SUNMatrix`, `SUNLinearSolver`, and `SUNNonlinearSolver` classes is given alongside the C documentation. For details on where the Fortran 2003 module (`.mod`) files and libraries are installed see [§17](#).

The Fortran 2003 interface modules were generated with SWIG Fortran [\[65\]](#), a fork of SWIG. Users who are interested in the SWIG code used in the generation process should contact the SUNDIALS development team.

Table 20.1: List of SUNDIALS Fortran 2003 interface modules

Class/Module	Fortran 2003 Module Name
SUNDIALS core	fsundials_core_mod
ARKODE	farkode_mod
ARKODE::ARKSTEP	farkode_arkstep_mod
ARKODE::ERKSTEP	farkode_erkstep_mod
ARKODE::MRISTEP	farkode_mrimestep_mod
ARKODE::SPRKSTEP	farkode_sprkstep_mod
ARKODE::LSRKSTEP	farkode_lsrkstep_mod
ARKODE::SPLITTINGSTEP	farkode_splittingstep_mod
ARKODE::FORCINGSTEP	farkode_forcingstep_mod
CVODE	fcvode_mod
CVODES	fcvodes_mod
IDA	fida_mod
IDAS	fidas_mod
KINSOL	fkinsol_mod
NVECTOR_CUDA	Not interfaced
NVECTOR_MANVECTOR	fnvector_manyvector_mod
NVECTOR_MPIMANVECTOR	fnvector_mpimanyvector_mod
NVECTOR_MPIPLUSX	fnvector_mpiplusx_mod
NVECTOR_OPENMP	fnvector_openmp_mod
NVECTOR_PARALLEL	fnvector_parallel_mod
NVECTOR_PARHYP	Not interfaced
NVECTOR_PETSC	Not interfaced
NVECTOR_PTHREADS	fnvector_pthreads_mod
NVECTOR_RAJA	Not interfaced
NVECTOR_SERIAL	fnvector_serial_mod
NVECTOR_SYCL	Not interfaced
SUNADAPTCONTROLLER_IMEXGUS	fsunadaptcontroller_imexgus_mod
SUNADAPTCONTROLLER_SODERLIND	fsunadaptcontroller_soderlind_mod
SUNADAPTCONTROLLER_MRIHTOL	fsunadaptcontroller_mrihtol_mod
SUNADJOINTCHECKPOINTSHEME_FIXED	fsunadjointcheckpointscheme_fixed_mod
SUNDOMEIGEST_ARNOLDI	fsundomeigest_arnoldi_mod
SUNDOMEIGEST_POWER	fsundomeigest_power_mod
SUNLINSOL_BAND	fsunlinsol_band_mod
SUNLINSOL_DENSE	fsunlinsol_dense_mod
SUNLINSOL_KLU	fsunlinsol_klu_mod
SUNLINSOL_LAPACKBAND	Not interfaced
SUNLINSOL_LAPACKDENSE	Not interfaced
SUNLINSOL_MAGMADENSE	Not interfaced
SUNLINSOL_ONEMKLDENSE	Not interfaced
SUNLINSOL_PCG	fsunlinsol_pcg_mod
SUNLINSOL_SLUDIST	Not interfaced
SUNLINSOL_SLUMT	Not interfaced
SUNLINSOL_SPBCGS	fsunlinsol_spbcgs_mod
SUNLINSOL_SPGMR	fsunlinsol_spgmr_mod
SUNLINSOL_SPTFQMR	fsunlinsol_sptfqmr_mod
SUNMATRIX_BAND	fsunmatrix_band_mod
SUNMATRIX_DENSE	fsunmatrix_dense_mod
SUNMATRIX_MAGMADENSE	Not interfaced

continues on next page

Table 20.1 – continued from previous page

Class/Module	Fortran 2003 Module Name
SUNMATRIX_ONEMKLDENSE	Not interfaced
SUNMATRIX_SPARSE	fsunmatrix_sparse_mod
SUNNONLINSOL_FIXEDPOINT	fsunnonlinsol_fixedpoint_mod
SUNNONLINSOL_NEWTON	fsunnonlinsol_newton_mod
SUNNONLINSOL_PETSCSNES	Not interfaced

### 20.1.1 Installation

The installation procedure for the Fortran interfaces is the same as for the C/C++ core of SUNDIALS, refer to §17. The CMake option to turn on the Fortran interfaces in a SUNDIALS build is `SUNDIALS_ENABLE_FORTRAN`. The Spack variant is `+fortran`.

### 20.1.2 Important notes on portability

The SUNDIALS Fortran 2003 interface *should* be compatible with any compiler supporting the Fortran 2003 ISO standard.

Upon compilation of SUNDIALS, Fortran module (.mod) files are generated for each Fortran 2003 interface. These files are highly compiler specific, and thus it is almost always necessary to compile a consuming application with the same compiler that was used to generate the modules.

## 20.2 Data Types

Generally, the Fortran 2003 type that is equivalent to the C type is what one would expect. Primitive types map to the `iso_c_binding` type equivalent. SUNDIALS classes map to a Fortran derived type. However, the handling of pointer types is not always clear as they can depend on the parameter direction. Table 20.2 presents a summary of the type equivalencies with the parameter direction in mind.

#### Warning

Currently, the Fortran 2003 interfaces are only compatible with SUNDIALS builds where the `sunrealtype` is double-precision.

Changed in version 7.1.0: The Fortran interfaces can now be built with 32-bit `sunindextype` in addition to 64-bit `sunindextype`.

Table 20.2: C/Fortran-2003 Equivalent Types. T represents any type.

C Type	Parameter Direction	Fortran 2003 type
SUNComm	in, inout, out, return	integer(c_int)
SUNErrCode	in, inout, out, return	integer(c_int)
double	in, inout, out, return	real(c_double)
int	in, inout, out, return	integer(c_int)
long	in, inout, out, return	integer(c_long)
sunbooleantype	in, inout, out, return	integer(c_int)
sunrealtype	in, inout, out, return	real(c_double)

continues on next page

Table 20.2 – continued from previous page

C Type	Parameter Direction	Fortran 2003 type
sunindextype	in, inout, out, return	integer(c_long)
double*	in, inout, out	real(c_double), dimension(*)
double*	return	real(c_double), pointer, dimension(:)
int*	in, inout, out	real(c_int), dimension(*)
int*	return	real(c_int), pointer, dimension(:)
long*	in, inout, out	real(c_long), dimension(*)
long*	return	real(c_long), pointer, dimension(:)
sunrealtype*	in, inout, out	real(c_double), dimension(*)
sunrealtype*	return	real(c_double), pointer, dimension(:)
sunindextype*	in, inout, out	real(c_long), dimension(*)
sunindextype*	return	real(c_long), pointer, dimension(:)
sunrealtype[]	in, inout, out	real(c_double), dimension(*)
sunindextype[]	in, inout, out	integer(c_long), dimension(*)
SUNAdaptController	in, inout, out	type(SUNAdaptController)
SUNAdaptController	return	type(SUNAdaptController), pointer
SUNAdjointCheckpointScheme	in, inout, out, return	type(c_ptr)
SUNAdjointStepper	in, inout, out, return	type(c_ptr)
SUNDomEigEstimator	in, inout, out	type(SUNDomEigEstimator)
SUNDomEigEstimator	return	type(SUNDomEigEstimator), pointer
SUNMatrix	in, inout, out	type(SUNMatrix)
SUNMatrix	return	type(SUNMatrix), pointer
SUNLinearSolver	in, inout, out	type(SUNLinearSolver)
SUNLinearSolver	return	type(SUNLinearSolver), pointer
SUNNonlinearSolver	in, inout, out	type(SUNNonlinearSolver)
SUNNonlinearSolver	return	type(SUNNonlinearSolver), pointer
N_Vector	in, inout, out	type(N_Vector)
N_Vector	return	type(N_Vector), pointer
FILE*	in, inout, out, return	type(c_ptr)
void*	in, inout, out, return	type(c_ptr)
T**	in, inout, out, return	type(c_ptr)
T***	in, inout, out, return	type(c_ptr)
T****	in, inout, out, return	type(c_ptr)

## 20.3 Notable Fortran/C usage differences

While the Fortran 2003 interface to SUNDIALS closely follows the C API, some differences are inevitable due to the differences between Fortran and C. In this section, we note the most critical differences. Additionally, §20.2 discusses equivalencies of data types in the two languages.

### 20.3.1 Creating generic SUNDIALS objects

In the C API a SUNDIALS class, such as an *N\_Vector*, is actually a pointer to an underlying C struct. However, in the Fortran 2003 interface, the derived type is bound to the C struct, not the pointer to the struct. For example, `type(N_Vector)` is bound to the C struct `_generic_N_Vector` not the `N_Vector` type. The consequence of this is that creating and declaring SUNDIALS objects in Fortran is nuanced. This is illustrated in the code snippets below:

C code:



```
N_Vector x;
x = N_VNew_Serial(N, sunctx);
```

Fortran code:

```
type(N_Vector), pointer :: x
x => FN_VNew_Serial(N, sunctx)
```

Note that in the Fortran declaration, the vector is a `type(N_Vector)`, `pointer`, and that the pointer assignment operator is then used.

### 20.3.2 Arrays and pointers

Unlike in the C API, in the Fortran 2003 interface, arrays and pointers are treated differently when they are return values versus arguments to a function. Additionally, pointers which are meant to be out parameters, not arrays, in the C API must still be declared as a rank-1 array in Fortran. The reason for this is partially due to the Fortran 2003 standard for C bindings, and partially due to the tool used to generate the interfaces. Regardless, the code snippets below illustrate the differences.

C code:

```
N_Vector x;
sunrealtype* xdata;
long int leniw, lenrw;

/* create a new serial vector */
x = N_VNew_Serial(N, sunctx);

/* capturing a returned array/pointer */
xdata = N_VGetArrayPointer(x)

/* passing array/pointer to a function */
N_VSetArrayPointer(xdata, x)

/* pointers that are out-parameters */
N_VSpace(x, &leniw, &lenrw);
```

Fortran code:

```
type(N_Vector), pointer :: x
real(c_double), pointer :: xdataptr(:)
real(c_double)          :: xdata(N)
integer(c_long)         :: leniw(1), lenrw(1)

! create a new serial vector
x => FN_VNew_Serial(x, sunctx)

! capturing a returned array/pointer
xdataptr => FN_VGetArrayPointer(x)

! passing array/pointer to a function
call FN_VSetArrayPointer(xdata, x)
```

(continues on next page)

(continued from previous page)

```
! pointers that are out-parameters
call FN_VSpace(x, leniw, lenrw)
```

### 20.3.3 Passing procedure pointers and user data

Since functions/subroutines passed to SUNDIALS will be called from within C code, the Fortran procedure must have the attribute `bind(C)`. Additionally, when providing them as arguments to a Fortran 2003 interface routine, it is required to convert a procedure's Fortran address to C with the Fortran intrinsic `c_funloc`.

Typically when passing user data to a SUNDIALS function, a user may simply cast some custom data structure as a `void*`. When using the Fortran 2003 interfaces, the same thing can be achieved. Note, the custom data structure *does not* have to be `bind(C)` since it is never accessed on the C side.

C code:

```
MyUserData *udata;
void *ccode_mem;

ierr = CCodeSetUserData(ccode_mem, udata);
```

Fortran code:

```
type(MyUserData), target :: udata
type(c_ptr)              :: ccode_mem

ierr = FCCodeSetUserData(ccode_mem, c_loc(udata))
```

On the other hand, Fortran users may instead choose to store problem-specific data, e.g. problem parameters, within modules, and thus do not need the SUNDIALS-provided `user_data` pointers to pass such data back to user-supplied functions. These users should supply the `c_null_ptr` input for `user_data` arguments to the relevant SUNDIALS functions.

### 20.3.4 Passing NULL to optional parameters

In the SUNDIALS C API some functions have optional parameters that a caller can pass as `NULL`. If the optional parameter is of a type that is equivalent to a Fortran `type(c_ptr)` (see §20.2), then a Fortran user can pass the intrinsic `c_null_ptr`. However, if the optional parameter is of a type that is not equivalent to `type(c_ptr)`, then a caller must provide a Fortran pointer that is dissociated. This is demonstrated in the code example below.

C code:

```
SUNLinearSolver LS;
N_Vector x, b;

/* SUNLinSolSolve expects a SUNMatrix or NULL as the second parameter. */
ierr = SUNLinSolSolve(LS, NULL, x, b);
```

Fortran code:

```
type(SUNLinearSolver), pointer :: LS
type(SUNMatrix), pointer      :: A
type(N_Vector), pointer       :: x, b
```

(continues on next page)

(continued from previous page)

```

! Disassociate A
A => null()

! SUNLinSolSolve expects a type(SUNMatrix), pointer as the second parameter.
! Therefore, we cannot pass a c_null_ptr, rather we pass a disassociated A.
ierr = FSUNLinSolSolve(LS, A, x, b)

```

### 20.3.5 Working with N\_Vector arrays

Arrays of *N\_Vector* objects are interfaced to Fortran 2003 as an opaque type (*c\_ptr*). As such, it is not possible to directly index an array of *N\_Vector* objects returned by the *N\_Vector* “VectorArray” operations, or packages with sensitivity capabilities (CVODES and IDAS). Instead, SUNDIALS provides a utility function *FN\_VGetVecAtIndexVectorArray* wrapping *N\_VGetVecAtIndexVectorArray()*. The example below demonstrates accessing a vector in a vector array.

C code:

```

N_Vector x;
N_Vector* vecs;

/* Create an array of N_Vectors */
vecs = N_VCloneVectorArray(count, x);

/* Fill each array with ones */
for (int i = 0; i < count; ++i)
    N_VConst(vecs[i], 1.0);

```

Fortran code:

```

type(N_Vector), pointer :: x, xi
type(c_ptr)             :: vecs

! Create an array of N_Vectors
vecs = FN_VCloneVectorArray(count, x)

! Fill each array with ones
do index = 0, count-1
    xi => FN_VGetVecAtIndexVectorArray(vecs, index)
    call FN_VConst(xi, 1.d0)
enddo

```

SUNDIALS also provides the functions *N\_VSetVecAtIndexVectorArray()* and *N\_VNewVectorArray()* for working with *N\_Vector* arrays, that have corresponding Fortran interfaces *FN\_VSetVecAtIndexVectorArray* and *FN\_VNewVectorArray*, respectively. These functions are particularly useful for users of the Fortran interface to the *NVECTOR\_MANYVECTOR* or *NVECTOR\_MPIMANYVECTOR* when creating the subvector array. Both of these functions along with *N\_VGetVecAtIndexVectorArray()* (wrapped as *FN\_VGetVecAtIndexVectorArray*) are further described in §8.1.1.

### 20.3.6 Providing file pointers

There are a few functions in the SUNDIALS C API which take a `FILE*` argument. Since there is no portable way to convert between a Fortran file descriptor and a C file pointer, SUNDIALS provides two utility functions for creating a `FILE*` and destroying it. These functions are defined in the module `fsundials_core_mod`.

*SUNErrCode* **SUNDIALSFileOpen**(const char \*filename, const char \*mode, FILE \*\*fp)

Deprecated since version 7.6.0: See [\*SUNFileOpen\(\)\*](#).

*SUNErrCode* **SUNFileOpen**(const char \*filename, const char \*mode, FILE \*\*fp)

The function allocates a `FILE*` by calling the C function `fopen` with the provided filename and I/O mode.

#### Parameters

- **filename** – the path to the file, that should have Fortran type `character(kind=C_CHAR, len=*)`. There are two special filenames: `stdout` and `stderr` – these two filenames will result in output going to the standard output file and standard error file, respectively.
- **mode** – the I/O mode to use for the file. This should have the Fortran type `character(kind=C_CHAR, len=*)`. The string begins with one of the following characters:
  - `r` to open a text file for reading
  - `r+` to open a text file for reading/writing
  - `w` to truncate a text file to zero length or create it for writing
  - `w+` to open a text file for reading/writing or create it if it does not exist
  - `a` to open a text file for appending, see documentation of `fopen` for your system/compiler
  - `a+` to open a text file for reading/appending, see documentation for `fopen` for your system/compiler
- **fp** – The `FILE*` that will be open when the function returns. This should be a *type(c\_ptr)* in the Fortran.

#### Returns

A *SUNErrCode*

Usage example:

```
type(c_ptr) :: fp

! Open up the file output.log for writing
ierr = FSUNFileOpen("output.log", "w+", fp)

! The C function ARKStepPrintMem takes void* arkode_mem and FILE* fp as arguments
call FARKStepPrintMem(arkode_mem, fp)

! Close the file
ierr = FSUNDIALSFileClose(fp)
```

Added in version 7.6.0: Replaces `SUNDIALSFileOpen`

*SUNErrCode* **SUNDIALSFileClose**(FILE \*\*fp)

Deprecated since version 7.6.0: See [\*SUNFileClose\(\)\*](#)

*SUNErrCode* **SUNFileClose**(FILE \*\*fp)

The function deallocates a C `FILE*` by calling the C function `fclose` with the provided pointer.

#### Parameters

- **fp** – the C FILE\* that was previously obtained from fopen. This should have the Fortran type type(c\_ptr). Note that if either stdout or stderr were opened using [SUN-FileOpen\(\)](#)

#### Returns

A [SUNErrCode](#)

Added in version 7.6.0: Replaces SUNDIALSFileClose

## 20.4 Common Issues

In this subsection, we list some common issues users run into when using the Fortran interfaces.

### Strange Segmentation Fault in User-Supplied Functions

One common issue we have seen trip up users (and even ourselves) has the symptom of segmentation fault in a user-supplied function (such as the RHS) when trying to use one of the callback arguments. For example, in the following RHS function, we will get a segfault on line 21:

```

1  integer(c_int) function ff(t, yvec, ydotvec, user_data) &
2      result(ierr) bind(C)
3
4      use, intrinsic :: iso_c_binding
5      use fsundials_nvector_mod
6      implicit none
7
8      real(c_double) :: t ! <===== Missing value attribute
9      type(N_Vector) :: yvec
10     type(N_Vector) :: ydotvec
11     type(c_ptr)    :: user_data
12
13     real(c_double) :: e
14     real(c_double) :: u, v
15     real(c_double) :: tmp1, tmp2
16     real(c_double), pointer :: yarr(:)
17     real(c_double), pointer :: ydotarr(:)
18
19     ! get N_Vector data arrays
20     yarr => FN_VGetArrayPointer(yvec)
21     ydotarr => FN_VGetArrayPointer(ydotvec) ! <===== SEGFAULTS HERE
22
23     ! extract variables
24     u = yarr(1)
25     v = yarr(2)
26
27     ! fill in the RHS function:
28     ! [0  0]*[(-1+u^2-r(t))/(2*u)] + [          0          ]
29     ! [e -1] [(-2+v^2-s(t))/(2*v)]   [sdot(t)/(2*vtrue(t))]]
30     tmp1 = (-ONE+u*u-r(t))/(TWO*u)
31     tmp2 = (-TWO+v*v-s(t))/(TWO*v)
32     ydotarr(1) = ZERO
33     ydotarr(2) = e*tmp1 - tmp2 + sdot(t)/(TWO*vtrue(t))
34
35     ! return success

```

(continues on next page)

(continued from previous page)

```
36   ierr = 0
37   return
38
39 end function
```

The subtle bug in the code causing the segfault is on line 8. It should read `real(c_double), value :: t` instead of `real(c_double) :: t` (notice the `value` attribute). Fundamental types that are passed by value in C need the `value` attribute.

# Chapter 21

## Python

### Warning

sundials4py is in beta and is subject to breaking changes. We welcome feedback on this new feature of SUNDIALS. Please report issues via the [SUNDIALS GitHub repository](#).

`sundials4py` provides official (supported by the SUNDIALS team) Python bindings to much of the SUNDIALS library, allowing you to use SUNDIALS directly from Python.

The bindings are built using `nanobind` and `litgen` and are designed to be easy to use from Python in conjunction with ubiquitous libraries in the Python scientific computing and machine learning ecosystems. To that end, `sundials4py` supports:

- Python’s automatic memory management
- Python definitions of user-supplied callback functions
- Zero-copy exchange of arrays (CPU and Device) through DLPack protocol and `numpy`’s `ndarray`

### Note

`sundials4py` requires Python 3.12+

The Python User Guide focuses on specific aspects of using SUNDIALS from Python and assumes the user is familiar with SUNDIALS. New SUNDIALS users should first read the *General User Guide* to understand the features and usage of SUNDIALS packages.

### 21.1 Using `sundials4py`

At a high level, using SUNDIALS from Python via `sundials4py` looks a lot like using SUNDIALS from C or C++. Below we overview using `sundials4py` and discuss the few notable differences.

### 21.1.1 Installation

You can install sundials4py directly from [PyPI](#) using pip:

```
pip install sundials4py
```

You can also install sundials4py from git:

```
pip install git+https://github.com/LLNL/sundials.git
```

The default build of sundials4py that is distributed as a binary wheel uses double precision real types and 64-bit indices. To install SUNDIALS with different precisions and index sizes, you can build from source wheels instead of using the pre-built binary wheels. When building from source wheels instead of binary wheels, you can customize the SUNDIALS precision (real type) and index type at build time by passing the CMake arguments in an environment variable when running pip. For example:

```
export CMAKE_ARGS="-DSUNDIALS_PRECISION=SINGLE -DSUNDIALS_INDEX_SIZE=64"
pip install sundials4py --no-binary=sundials4py
```

Other SUNDIALS options can also be accessed in this way. Review §17.3 for more information on the available options.

### 21.1.2 Modules

After installation, you can import the sundials4py module with

```
import sundials4py
```

which includes the following submodules (which may also be individually imported) for accessing specific SUNDIALS features:

- `sundials4py.core` contains all the shared SUNDIALS classes and functions as well as many of the native SUNDIALS class implementations:
  - `NVector`: serial and many-vector
  - `SUNMatix`: band, dense, and sparse
  - `SUNLinearSover`: band, dense, PCG, SPBCGS, SPFGMR, SPGMR, and SPTFQMR
  - `SUNNonlinearSolver`: fixed-point and Newton
  - `SUNAdaptController`: Soderlind, ImEx-Gus, and MRI H-Tol
  - `SUNDomEigEstimator`: Power
  - `SUNAdjointCheckPointScheme`: Fixed
- `sundials4py.arkode` contains all of the ARKODE specific classes and functions
- `sundials4py.cvodes` contains all of the CVODES specific classes and functions
- `sundials4py.idas` contains all of the IDAS specific classes and functions
- `sundials4py.kinsol` contains all of the KINSOL specific classes and functions

CVODE and IDA do not have modules because CVODES and IDAS provide all of the same capabilities plus continuous forward and adjoint sensitivity analysis.



**Note**

Not all SUNDIALS features are supported by the Python interfaces. In particular, third-party libraries are not yet supported.

### 21.1.3 Example Usage

We now consider a simple CVODE example to illustrate using sundials4py and highlight some of the differences to using SUNDIALS from C/C++. The items highlighted below similarly apply to using other SUNDIALS packages. For more information on usage differences, continue to the [next section](#). Additional examples can be found in the `examples/python` directory of the [SUNDIALS GitHub repository](#).

This example demonstrates how to use CVODES to solve the Lotka-Volterra equations, a model of predator-prey dynamics in ecology, given by

$$\begin{aligned}u' &= p_0 u - p_1 uv \\v' &= -p_2 v + p_3 uv\end{aligned}$$

where  $u$  is the prey population,  $v$  is the predator population,  $p_0$  is prey birth rate,  $p_1$  is the predation rate,  $p_2$  is the predator death rate, and  $p_3$  is predator growth rate from predation. We use the parameters  $p = [1.5, 1.0, 3.0, 1.0]$ , initial condition  $y(0) = [1.0, 1.0]$ , and integration interval  $t \in [0, 10]$ .

```

1  import numpy as np
2  import sys
3  import matplotlib.pyplot as plt
4  from sundials4py.core import * # Always import the core submodule
5  from sundials4py.cvodes import * # Import the desired SUNDIALS package
6
7
8  class LotkaVolterraODE:
9      """
10         Encapsulates the Lotka-Volterra ODE problem.
11
12         This class defines the ODE system and provides the functions that CVODE needs to
13         evolve it in time: the right-hand side (RHS) function and the Jacobian.
14         """
15
16     def __init__(self, p):
17         """
18         Initialize with model parameters.
19
20         Args:
21             p: list or array of 4 parameters [p_0, p_1, p_2, p_3]
22         """
23         self.p = np.array(p, dtype=sunrealtype)
24         self.NEQ = 2 # Number of equations in the system
25
26     def set_init_cond(self, yvec):
27         """
28         Set the initial conditions in the solution vector.
29
30         Args:

```

(continues on next page)

(continued from previous page)

```

31         yvec: N_Vector to store initial values
32
33     Returns:
34         0 on success
35     """
36     y = N_VGetArrayPointer(yvec) # Returns a numpy ndarray view of the data
37     y[0] = 1.0 # Initial prey population
38     y[1] = 1.0 # Initial predator population
39     return 0
40
41     def rhs(self, t, yvec, ydotvec, _):
42         """
43         Right-hand side function: computes  $dy/dt = f(t, y)$ .
44
45         Args:
46             t: current time (not used in this autonomous system)
47             yvec: N_Vector with the current solution values  $y = [u, v]$ 
48             ydotvec: N_Vector output vector for time derivatives  $f = [du/dt, dv/dt]$ 
49             _: user data pointer MUST NOT be used in Python
50
51         Returns:
52             0 on success
53             positive value for a recoverable error (integrator will retry the step)
54             negative value for an unrecoverable error (integration will halt)
55         """
56         p = self.p
57         y = N_VGetArrayPointer(yvec) # Returns a numpy ndarray view of the data
58         ydot = N_VGetArrayPointer(ydotvec)
59
60         # Compute the derivatives
61         ydot[0] = p[0] * y[0] - p[1] * y[0] * y[1] # du/dt: prey dynamics
62         ydot[1] = -p[2] * y[1] + p[3] * y[0] * y[1] # dv/dt: predator dynamics
63         return 0
64
65     def jac(self, t, yvec, fyvec, J, _, tmp1, tmp2, tmp3):
66         """
67         Jacobian function: computes  $J = df/dy$ .
68
69         Args:
70             t: current time
71             yvec: N_Vector with the current solution values
72             fyvec: N_Vector with the current RHS values (not used here)
73             J: SUNmatrix to store the output Jacobian matrix
74             _: user data pointer MUST NOT be used in Python
75             tmp1, tmp2, tmp3: temporary workspace N_Vectors (not used here)
76
77         Returns:
78             0 on success
79             positive value for a recoverable error (integrator will retry the step)
80             negative value for an unrecoverable error (integration will halt)
81         """
82         p = self.p

```

(continues on next page)

(continued from previous page)

```

83     y = N_VGetArrayPointer(yvec) # Returns a numpy ndarray view of the data
84     Jdata = SUNDenseMatrix_Data(J) # Returns a numpy ndarray view of the data
85
86     # Compute partial derivatives
87     # J[0,0] = d(du/dt)/du, J[0,1] = d(du/dt)/dv
88     Jdata[0, 0] = p[0] - p[1] * y[1]
89     Jdata[0, 1] = -p[1] * y[0]
90
91     # J[1,0] = d(dv/dt)/du, J[1,1] = d(dv/dt)/dv
92     Jdata[1, 0] = p[3] * y[1]
93     Jdata[1, 1] = -p[2] + p[3] * y[0]
94     return 0
95
96
97 def main():
98     """
99     Main function demonstrating the SUNDIALS/CVODE workflow.
100
101     A typical workflow for solving an ODE with CVODE is:
102     1. Create the SUNDIALS context
103     2. Create the solution vector and set initial conditions
104     3. Create and initialize CVODE
105     4. Configure CVODE (set tolerances, linear solver, Jacobian, etc.)
106     5. Advance the solution in time
107     6. Retrieve CVODE statistics
108     7. Destroy objects and free memory (handled automatically in Python)
109     """
110
111     # -----
112     # Step 1: Create the SUNDIALS context
113     # -----
114
115     # The context provides support for features such as error handling, logging, and
116     # profiling and is required to construct all SUNDIALS objects. SUN_COMM_NULL is used
117     # for serial (non-parallel) problems.
118
119     # In C, sunctx is an output parameter (return-by-pointer):
120     # SUNContext_Create(SUN_COMM_NULL, &sunctx). In Python, it is returned in a tuple
121     # where the first entry is the function return value and the second is the context.
122     status, sunctx = SUNContext_Create(SUN_COMM_NULL)
123     assert status == SUN_SUCCESS
124
125     # -----
126     # Step 2: Set up the ODE problem
127     # -----
128
129     # Create the ODE problem with parameters p = [1.5, 1.0, 3.0, 1.0]
130     params = [1.5, 1.0, 3.0, 1.0]
131     ode = LotkaVolterraODE(params)
132
133     # Create a serial N_Vector to hold the solution
134     y = N_VNew_Serial(ode.NEQ, sunctx)

```

(continues on next page)

(continued from previous page)

```

135 assert y is not None
136
137 # Set initial conditions: y(0) = [1.0, 1.0]
138 ode.set_init_cond(y)
139
140 # -----
141 # Step 3: Create and initialize CVODE
142 # -----
143
144 # Create a CVODE instance using Backward Differentiation Formulas (BDF)
145 ccode = CCodeCreate(CV_BDF, sunctx)
146 assert ccode is not None
147
148 # Initialize CVODE with the RHS function, initial time (t0=0.0), and initial state
149 status = CCodeInit(ccode.get(), ode.rhs, 0.0, y) # access CVODE memory with .get()
150 assert status == CV_SUCCESS
151
152 # -----
153 # Step 4: Set CVODE options (tolerances, linear solver, etc.)
154 # -----
155
156 # CVODE will adapt the step size and method order so the estimated local error meets
157 # the user defined tolerances.
158 reltol = 1e-6 # Relative tolerance
159 abstol = 1e-10 # Absolute tolerance
160 status = CCodeSetTolerances(ccode.get(), reltol, abstol)
161 assert status == CV_SUCCESS
162
163 # CVODE defaults to using a Newton iteration to solve nonlinear systems at each time
164 # step which requires solving a linear system each iteration. For small problems, a
165 # dense direct solver is efficient and easy to use.
166
167 # Create a dense matrix (NEQ x NEQ) to store the Jacobian
168 A = SUNDenseMatrix(ode.NEQ, ode.NEQ, sunctx)
169 assert A is not None
170
171 # Create a dense linear solver that works with the matrix A and vector y
172 LS = SUNLinSol_Dense(y, A, sunctx)
173 assert LS is not None
174
175 # Attach the linear solver to CVODE
176 status = CCodeSetLinearSolver(ccode.get(), LS, A)
177 assert status == CV_SUCCESS
178
179 # Providing an analytical Jacobian can significantly improve performance. If not
180 # provided, CVODE will approximate it using finite differences.
181 status = CCodeSetJacFn(ccode.get(), ode.jac)
182 assert status == CV_SUCCESS
183
184 # -----
185 # Step 5: Advance the ODE in time
186 # -----

```

(continues on next page)

(continued from previous page)

```

187
188 # Set the current output time and get access to the underlying array data for output
189 tret = 0.0
190 yarr = N_VGetArrayPointer(y) # Returns a numpy ndarray view of the data
191
192 # Print header with problem information
193 print("\nLotka-Volterra ODE (CVODE):")
194 print(f"    initial condition, y0 = [1.0, 1.0]")
195 print(f"    parameters = {params}")
196 print(f"    reltol = {reltol}, abstol = {abstol}\n")
197 print("        t            u            v")
198 print("    -----")
199 print(f" {tret:10.6f} {yarr[0]:10.6f} {yarr[1]:10.6f}")
200
201 # Lists to store solution for plotting
202 t_vals = [tret]
203 u_vals = [yarr[0]]
204 v_vals = [yarr[1]]
205
206 # Integrate from t=0 to t=10, outputting every 0.05 time units using CV_NORMAL mode.
207 # In normal mode, CVODE will take internal steps until it has reached or just passed
208 # the output time and then return a time interpolated solution at the output time.
209 dtout = 0.05
210 iout = 0
211 while tret < 10.0:
212     # Advance the system and return the solution at requested output time
213     status, tret = CVode(cvode.get(), tret + dtout, y, CV_NORMAL)
214     assert status == CV_SUCCESS
215
216     # Store solution for plotting (every output)
217     t_vals.append(tret)
218     u_vals.append(yarr[0])
219     v_vals.append(yarr[1])
220
221     # Print every 10th output (every 1.0 time unit) to keep the output readable
222     iout += 1
223     if iout % 10 == 0 or tret >= 10.0:
224         print(f" {tret:10.6f} {yarr[0]:10.6f} {yarr[1]:10.6f}")
225 print("    -----")
226
227 # -----
228 # Step 6: Get CVODE statistics
229 # -----
230
231 # CVODE provides various statistics about the integration that can help assess
232 # performance and diagnose issues. These include values such as the step counts,
233 # function evaluations, and information about the linear solver.
234
235 status, nst = CVodeGetNumSteps(cvode.get())
236 assert status == CV_SUCCESS
237
238 status, netf = CVodeGetNumErrTestFails(cvode.get())

```

(continues on next page)

(continued from previous page)

```

239     assert status == CV_SUCCESS
240
241     status, nfe = CNodeGetNumRhsEvals(cnode.get())
242     assert status == CV_SUCCESS
243
244     status, nsetups = CNodeGetNumLinSolvSetups(cnode.get())
245     assert status == CV_SUCCESS
246
247     status, nje = CNodeGetNumJacEvals(cnode.get())
248     assert status == CV_SUCCESS
249
250     # For C functions with multiple output parameters (return-by-pointer),
251     # CNodeGetNonlinSolvStats(cnode_mem, &nni, &ncfn), the first element is always the
252     # function return value (error code) and the subsequent elements follow the same
253     # ordering as in the C function signature.
254     status, nni, ncfn = CNodeGetNonlinSolvStats(cnode.get())
255     assert status == CV_SUCCESS
256
257     print("\nIntegrator Statistics:")
258     print(f"    Number of steps taken                = {nst}")
259     print(f"    Number of error test fails            = {netf}")
260     print(f"    Number of RHS evaluations             = {nfe}")
261     print(f"    Number of linear solver setups        = {nsetups}")
262     print(f"    Number of Jacobian evaluations        = {nje}")
263     print(f"    Number of nonlinear solver iterations = {nni}")
264     print(f"    Number of nonlinear convergence failures = {ncfn}")
265
266     # -----
267     # Plot the solution
268     # -----
269
270     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
271
272     # Left plot: Time series of prey and predator populations
273     ax1.plot(t_vals, u_vals, "b-", label="Prey (u)")
274     ax1.plot(t_vals, v_vals, "r--", label="Predator (v)")
275     ax1.set_xlabel("Time")
276     ax1.set_ylabel("Population")
277     ax1.set_title("Lotka-Volterra: Population vs Time")
278     ax1.legend()
279     ax1.grid(alpha=0.3)
280
281     # Right plot: Phase portrait (predator vs prey)
282     ax2.plot(u_vals, v_vals, "g-")
283     ax2.plot(u_vals[0], v_vals[0], "ko", label="Start")
284     ax2.plot(u_vals[-1], v_vals[-1], "ks", label="End")
285     ax2.set_xlabel("Prey (u)")
286     ax2.set_ylabel("Predator (v)")
287     ax2.set_title("Phase Portrait")
288     ax2.legend()
289     ax2.grid(alpha=0.3)
290

```

(continues on next page)

(continued from previous page)

```

291     # Display the plot if not running in test mode
292     if "pytest" not in sys.modules:
293         plt.tight_layout()
294         plt.show()
295     else:
296         plt.close("all")
297
298
299 if __name__ == "__main__":
300     main()

```

### 21.1.4 Usage Differences

While sundials4py closely follows the C API, some differences are inevitable due to the differences between Python and C as well as the requirements of the code generation tool used to create the bindings. In this section, we note the most critical differences.

#### 21.1.4.1 View Classes and Memory Management

sundials4py provides natural usage of SUNDIALS objects with object lifetimes managed by the Python garbage collection as with any other Python object. There is only one caveat, the SUNDIALS integrator/solver `void*` objects (those returned by ARKODE, CVODES, IDAS, and KINSOL `Create` constructors) are wrapped in “View” classes (behind the scenes) for compatibility with nanobind. These view objects cannot be implicitly converted to the underlying `void*`. As such, when calling a function which operates on these `void*` objects, one must extract the `void*` “capsule” from the view object by calling the view’s `get` method.

```

# Create CVODE object (returns void* in C)
ccode = CCodeCreate(CV_BDF, sunctx)

# Notice we need to call ccode.get()
status = CCodeInit(ccode.get(), ode_problem.f, T0, y)

```

#### 21.1.4.2 Return-by-Pointer Parameters

Functions that return values via pointer arguments in the C API are mapped to Python functions that return a tuple where the **first element** is the function’s return value (typically an error code) and **subsequent elements** are the values that would be returned via pointer arguments in C, in the same order as the C function signature.

##### Example 1: Single Return-by-Pointer Value

C:

```

int retval;
long int numsteps;
retval = CCodeGetNumSteps(ccode_mem, &numsteps);
printf("Number of steps: %ld\n", numsteps);

```

Python:

```

retval, numsteps = CCodeGetNumSteps(ccode_mem.get())
print(f"Number of steps: {numsteps}")

```

**Example 2: Multiple Return-by-Pointer Values****C:**

```
int retval;
long int nni, ncnf;
retval = CNodeGetNonlinSolvStats(cnode_mem, &nni, &ncnf)
printf("Nonlinear iterations: %ld, Nonlinear convergence fails: %ld\n", nni, ncnf);
```

**Python:**

```
retval, nni, ncnf = CNodeGetNonlinSolvStats(cnode_mem.get())
print(f"Nonlinear iterations: {nni}, Nonlinear convergence fails: {ncnf}");
```

**21.1.4.3 Arrays**

`N_Vector` objects in `sundials4py` are compatible with `numpy`'s `ndarray`. Each `N_Vector` can work on a `numpy` arrays without copies, and you can access and modify the underlying data directly using `N_VGetArrayPointer`, which returns a `numpy ndarray` view of the data.

SUNDIALS matrix types (dense, banded, sparse) are also exposed as Python objects that provide access to their underlying data as `numpy` arrays (e.g., via `SUNDenseMatrix_Data`).

Arrays of scalars (e.g., scaling factors passed to `N_VLinearCombination`) are also represented as `numpy` arrays.

**Example: Accessing and modifying an `N_Vector`**

```
y_nvec = N_VNew_Serial(10, sunctx)
y = N_VGetArrayPointer(y_nvec)
y[:] = np.linspace(0, 1, 10)  # Set values using numpy
```

**Example: Using a matrix as a `numpy` array**

```
mat = SUNDenseMatrix(3, 3, sunctx)
arr = SUNDenseMatrix_Data(mat)
arr[:] = np.eye(3)  # Set to identity matrix
```

This allows you to use `numpy` operations for vector and matrix data, and to pass `numpy` arrays to and from SUNDIALS routines efficiently and without unnecessary copies.

**21.1.4.4 User-Supplied Callback Functions**

SUNDIALS packages and several modules/classes require user-supplied callback functions to define problem-specific behavior, such as the right-hand side of an ODE or a nonlinear system function. In `sundials4py`, you can provide these as standard Python functions or lambdas.

The callback signatures follow the C API. As such, `N_Vector` arguments are passed as `N_Vector` objects and the underlying `ndarray` must be extracted in the user code. The only caveat is that return-by-pointer parameters are removed from the signature, and instead become return values (mirroring how return-by-pointer parameters for other functions are handled)

**Warning**



The C function signatures for most callbacks include a `void* user_data` argument. In Python, this argument must be present in the signature, but it should be ignored to avoid catastrophic errors. We recommend using `_` as the parameter name in the callback signature to indicate this argument is unused.

#### Example: ODE right-hand side for ARKStep

```
# The C signature is:
# int(sunrealtype t, N_Vector y, N_Vector ydot, void* user_data)
def rhs(t, y_nvector, ydot_nvector, _): # note _ in place of user_data
    # Compute ydot = f(t, y)
    y = N_VGetArrayPointer(y_nvector)
    ydot = N_VGetArrayPointer(ydot_nvector)
    ydot[:] = -y
    return 0

ark = ARKStepCreate(rhs, None, t0, y, sunctx)
```

#### Example: Nonlinear system for KINSOL

```
# The C signature is:
# int(N_Vector u, N_Vector g, void* user_data)
def fp_function(u_nvector, g_nvector, _): # note _ in place of user_data
    # Compute g = F(u)
    u = N_VGetArrayPointer(u_nvector)
    g = N_VGetArrayPointer(g_nvector)
    g[:] = u**2 - 1
    return 0

kin = KINCreate(sunctx)
KINInit(kin.get(), fp_function, u)
```

#### Example: ARKODE LSRKStep dominant eigenvalue estimation function with return-by-pointer parameters

```
# The C signature is:
# int(sunrealtype t, N_Vector y, N_Vector fn,
#     sunrealtype lambdaR, sunrealtype lambdaI,
#     void* user_data,
#     N_Vector temp1, N_Vector temp2, N_Vector temp3)
def dom_eig(t, yvec, fnvec, _, temp1, temp2, temp3): # note the _ in place of user_data
    lambdaR = L
    lambdaI = 0.0
    # lambdaR and lambdaI should be returned in the order that they appear
    # as parameters in the C API and follow the error code to return
    return 0, lambdaR, lambdaI
```

#### 21.1.4.5 Error Codes

The named `SUN_ERR_*` code constants are not available in Python. However, all negative values of `SUNErrCode` are still errors, zero is success, and positive values are warnings. As such, users can call `SUNGetErrMsg` from Python with the returned `SUNErrCode` to get further information about an error.

## 21.2 core Submodule

### 21.2.1 Classes

A submodule of ‘sundials4py’

```
class sundials4py.core.FILE
```

```
class sundials4py.core.N_Vector_ID(*values)
```

```
    SUNDIALS_NVEC_CUDA = 6
```

```
    SUNDIALS_NVEC_CUSTOM = 16
```

```
    SUNDIALS_NVEC_HIP = 7
```

```
    SUNDIALS_NVEC_KOKKOS = 10
```

```
    SUNDIALS_NVEC_MANYVECTOR = 13
```

```
    SUNDIALS_NVEC_MPIMANYVECTOR = 14
```

```
    SUNDIALS_NVEC_MPIPLUSX = 15
```

```
    SUNDIALS_NVEC_OPENMP = 2
```

```
    SUNDIALS_NVEC_OPENMPDEV = 11
```

```
    SUNDIALS_NVEC_PARALLEL = 1
```

```
    SUNDIALS_NVEC_PARHYP = 4
```

```
    SUNDIALS_NVEC_PETSC = 5
```

```
    SUNDIALS_NVEC_PTHREADS = 3
```

```
    SUNDIALS_NVEC_RAJA = 9
```

```
    SUNDIALS_NVEC_SERIAL = 0
```

```
    SUNDIALS_NVEC_SYCL = 8
```

```
    SUNDIALS_NVEC_TRILINOS = 12
```

```
class sundials4py.core.SUNAdaptControllerContent_MRIHTol_(*args, **kwargs)
```

```
class sundials4py.core.SUNAdaptController_Type(*values)
```

```
    SUN_ADAPTCONTROLLER_H = 1
```

```
    SUN_ADAPTCONTROLLER_MRI_H_TOL = 2
```

```
    SUN_ADAPTCONTROLLER_NONE = 0
```

```
class sundials4py.core.SUNAdjointCheckpointScheme_
```

```
class sundials4py.core.SUNAdjointStepper_
```

```
class sundials4py.core.SUNContext_
```

```
class sundials4py.core.SUNDataIOMode(*values)
```

```
    SUNDATAIOMODE_INMEM = 0
```

```
class sundials4py.core.SUNDomEigEstimatorContent_Power_(*args, **kwargs)
```

```
class sundials4py.core.SUNDomEigEstimator_(*args, **kwargs)
```

An estimator is a structure with an implementation-dependent ‘content’ field, and a pointer to a structure of estimator operations corresponding to that implementation.

```
class sundials4py.core.SUNDomEigEstimator_Ops_(*args, **kwargs)
```

Structure containing function pointers to estimator operations

```
class sundials4py.core.SUNErrCode(*values)
```

```
    SUN_ERR_ADJOINT_STEPPERFAILED = -9977
```

```
    SUN_ERR_ADJOINT_STEPPERINVALIDSTOP = -9976
```

```
    SUN_ERR_ARG_CORRUPT = -9999
```

```
    SUN_ERR_ARG_DIMSMISMATCH = -9995
```

```
    SUN_ERR_ARG_INCOMPATIBLE = -9998
```

```
    SUN_ERR_ARG_OUTOFRANGE = -9997
```

```
    SUN_ERR_ARG_WRONGTYPE = -9996
```

```
    SUN_ERR_CHECKPOINT_MISMATCH = -9974
```

```
    SUN_ERR_CHECKPOINT_NOT_FOUND = -9975
```

```
    SUN_ERR_CORRUPT = -9993
```

```
    SUN_ERR_DATANODE_NODENOTFOUND = -9983
```

```
    SUN_ERR_DESTROY_FAIL = -9986
```

```
    SUN_ERR_EXT_FAIL = -9987
```

```
    SUN_ERR_FILE_OPEN = -9991
```

```
    SUN_ERR_GENERIC = -9994
```

```
    SUN_ERR_MALLOC_FAIL = -9988
```

```
    SUN_ERR_MAXIMUM = -1000
```

```
    SUN_ERR_MEM_FAIL = -9989
```

```
    SUN_ERR_MINIMUM = -10000
```

```
    SUN_ERR_MPI_FAIL = -9972
```

```
    SUN_ERR_NOT_IMPLEMENTED = -9985
```

```
    SUN_ERR_OP_FAIL = -9990
```

```
    SUN_ERR_OUTOFRANGE = -9992
```

```
SUN_ERR_PROFILER_MAPFULL = -9982
SUN_ERR_PROFILER_MAPGET = -9981
SUN_ERR_PROFILER_MAPINSERT = -9980
SUN_ERR_PROFILER_MAPKEYNOTFOUND = -9979
SUN_ERR_PROFILER_MAPSORT = -9978
SUN_ERR_SUNCTX_CORRUPT = -9973
SUN_ERR_UNKNOWN = -9970
SUN_ERR_UNREACHABLE = -9971
SUN_ERR_USER_FCN_FAIL = -9984
SUN_SUCCESS = 0
```

```
class sundials4py.core.SUNFullRhsMode(*values)
```

```
SUN_FULLRHS_END = 1
SUN_FULLRHS_OTHER = 2
SUN_FULLRHS_START = 0
```

```
class sundials4py.core.SUNGramSchmidtType(*values)
```

```
SUN_CLASSICAL_GS = 2
SUN_MODIFIED_GS = 1
```

```
class sundials4py.core.SUNLinearSolver_ID(*values)
```

```
SUNLINEARSOLVER_BAND = 0
SUNLINEARSOLVER_CUSOLVERSPP_BATCHQR = 12
SUNLINEARSOLVER_CUSTOM = 18
SUNLINEARSOLVER_DENSE = 1
SUNLINEARSOLVER_GINKGO = 15
SUNLINEARSOLVER_GINKGOBATCH = 16
SUNLINEARSOLVER_KLU = 2
SUNLINEARSOLVER_KOKKOSDENSE = 17
SUNLINEARSOLVER_LAPACKBAND = 3
SUNLINEARSOLVER_LAPACKDENSE = 4
SUNLINEARSOLVER_MAGMADENSE = 13
SUNLINEARSOLVER_ONEMKLDENSE = 14
SUNLINEARSOLVER_PCG = 5
```

```

SUNLINEARSOLVER_SPBCGS = 6
SUNLINEARSOLVER_SPGMR = 7
SUNLINEARSOLVER_SPTFQMR = 9
SUNLINEARSOLVER_SUPERLUDIST = 10
SUNLINEARSOLVER_SUPERLUMT = 11
class sundials4py.core.SUNLinearSolver_Type(*values)
    SUNLINEARSOLVER_DIRECT = 0
    SUNLINEARSOLVER_ITERATIVE = 1
    SUNLINEARSOLVER_MATRIX_EMBEDDED = 3
    SUNLINEARSOLVER_MATRIX_ITERATIVE = 2
class sundials4py.core.SUNLogLevel(*values)
    SUN_LOGLEVEL_ALL = -1
    SUN_LOGLEVEL_DEBUG = 4
    SUN_LOGLEVEL_ERROR = 1
    SUN_LOGLEVEL_INFO = 3
    SUN_LOGLEVEL_NONE = 0
    SUN_LOGLEVEL_WARNING = 2
class sundials4py.core.SUNLogger_
class sundials4py.core.SUNMatrix_ID(*values)
    SUNMATRIX_BAND = 3
    SUNMATRIX_CUSPARSE = 6
    SUNMATRIX_CUSTOM = 10
    SUNMATRIX_DENSE = 0
    SUNMATRIX_GINKGO = 7
    SUNMATRIX_GINKGOBATCH = 8
    SUNMATRIX_KOKKODENSE = 9
    SUNMATRIX_MAGMADENSE = 1
    SUNMATRIX_ONEMKLDENSE = 2
    SUNMATRIX_SLUNRLOC = 5
    SUNMATRIX_SPARSE = 4

```

```
class sundials4py.core.SUNMemoryHelper_(*args, **kwargs)
class sundials4py.core.SUNMemoryHelper_Ops_(*args, **kwargs)
class sundials4py.core.SUNMemoryType(*values)
    SUNMEMTYPE_DEVICE = 2
    SUNMEMTYPE_HOST = 0
    SUNMEMTYPE_PINNED = 1
    SUNMEMTYPE_UVM = 3
class sundials4py.core.SUNNonlinearSolver_Type(*values)
    SUNNONLINEARSOLVER_FIXEDPOINT = 1
    SUNNONLINEARSOLVER_ROOTFIND = 0
class sundials4py.core.SUNOutputFormat(*values)
    SUN_OUTPUTFORMAT_CSV = 1
    SUN_OUTPUTFORMAT_TABLE = 0
class sundials4py.core.SUNPrecType(*values)
    SUN_PREC_BOTH = 3
    SUN_PREC_LEFT = 1
    SUN_PREC_NONE = 0
    SUN_PREC_RIGHT = 2
class sundials4py.core.SUNProfiler_
class sundials4py.core.SUNStepper_
class sundials4py.core._N_VectorContent_ManyVector(*args, **kwargs)
class sundials4py.core._N_VectorContent_Serial(*args, **kwargs)
class sundials4py.core._SUNAdaptControllerContent_ImExGus(*args, **kwargs)
class sundials4py.core._SUNAdaptControllerContent_Soderlind(*args, **kwargs)
class sundials4py.core._SUNLinearSolverContent_Band(*args, **kwargs)
class sundials4py.core._SUNLinearSolverContent_Dense(*args, **kwargs)
class sundials4py.core._SUNLinearSolverContent_PCG(*args, **kwargs)
class sundials4py.core._SUNLinearSolverContent_SPBCGS(*args, **kwargs)
class sundials4py.core._SUNLinearSolverContent_SPFGMR(*args, **kwargs)
class sundials4py.core._SUNLinearSolverContent_SPGMR(*args, **kwargs)
class sundials4py.core._SUNLinearSolverContent_SPTFQMR(*args, **kwargs)
```

```

class sundials4py.core._SUNMatrixContent_Band(*args, **kwargs)
class sundials4py.core._SUNMatrixContent_Dense(*args, **kwargs)
class sundials4py.core._SUNMatrixContent_Sparse(*args, **kwargs)
class sundials4py.core._SUNNonlinearSolverContent_FixedPoint(*args, **kwargs)
class sundials4py.core._SUNNonlinearSolverContent_Newton(*args, **kwargs)
class sundials4py.core._generic_N_Vector(*args, **kwargs)
    A vector is a structure with an implementation-dependent 'content' field, and a pointer to a structure of vector
    operations corresponding to that implementation.
class sundials4py.core._generic_N_Vector_Ops(*args, **kwargs)
    Structure containing function pointers to vector operations
class sundials4py.core._generic_SUNAdaptController(*args, **kwargs)
    A SUNAdaptController is a structure with an implementation-dependent 'content' field, and a pointer to a struc-
    ture of operations corresponding to that implementation.
class sundials4py.core._generic_SUNAdaptController_Ops(*args, **kwargs)
    Structure containing function pointers to controller operations
class sundials4py.core._generic_SUNLinearSolver(*args, **kwargs)
    A linear solver is a structure with an implementation-dependent 'content' field, and a pointer to a structure of
    linear solver operations corresponding to that implementation.
class sundials4py.core._generic_SUNLinearSolver_Ops(*args, **kwargs)
    Structure containing function pointers to linear solver operations
class sundials4py.core._generic_SUNMatrix(*args, **kwargs)
    A matrix is a structure with an implementation-dependent 'content' field, and a pointer to a structure of matrix
    operations corresponding to that implementation.
class sundials4py.core._generic_SUNMatrix_Ops(*args, **kwargs)
    Structure containing function pointers to matrix operations
class sundials4py.core._generic_SUNNonlinearSolver(*args, **kwargs)
    A nonlinear solver is a structure with an implementation-dependent 'content' field, and a pointer to a structure
    of solver nonlinear solver operations corresponding to that implementation.
class sundials4py.core._generic_SUNNonlinearSolver_Ops(*args, **kwargs)
    Structure containing function pointers to nonlinear solver operations

```

### 21.2.2 Functions

```

sundials4py.core.N_VAbs(x: sundials4py.core._generic_N_Vector, z: sundials4py.core._generic_N_Vector) →
    None

```

See [N\\_VAbs\(\)](#).

```

sundials4py.core.N_VAddConst(x: sundials4py.core._generic_N_Vector, b: float, z:
    sundials4py.core._generic_N_Vector) → None

```

See [N\\_VAddConst\(\)](#).

`sundials4py.core.N_VClone(w: sundials4py.core._generic_N_Vector) → sundials4py.core._generic_N_Vector`  
See [`N\_VClone\(\)`](#).

`sundials4py.core.N_VCloneEmpty(w: sundials4py.core._generic_N_Vector) → sundials4py.core._generic_N_Vector`  
See [`N\_VCloneEmpty\(\)`](#).

`sundials4py.core.N_VCompare(c: float, x: sundials4py.core._generic_N_Vector, z: sundials4py.core._generic_N_Vector) → None`  
See [`N\_VCompare\(\)`](#).

`sundials4py.core.N_VConst(c: float, z: sundials4py.core._generic_N_Vector) → None`  
See [`N\_VConst\(\)`](#).

`sundials4py.core.N_VConstVectorArray(nvec: int, c: float, Z_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector]) → int`  
See [`N\_VConstVectorArray\(\)`](#).

`sundials4py.core.N_VConstrMask(c: sundials4py.core._generic_N_Vector, x: sundials4py.core._generic_N_Vector, m: sundials4py.core._generic_N_Vector) → int`  
See [`N\_VConstrMask\(\)`](#).

`sundials4py.core.N_VConstrMaskLocal(c: sundials4py.core._generic_N_Vector, x: sundials4py.core._generic_N_Vector, m: sundials4py.core._generic_N_Vector) → int`  
See [`N\_VConstrMaskLocal\(\)`](#).

`sundials4py.core.N_VDiv(x: sundials4py.core._generic_N_Vector, y: sundials4py.core._generic_N_Vector, z: sundials4py.core._generic_N_Vector) → None`  
See [`N\_VDiv\(\)`](#).

`sundials4py.core.N_VDotProd(x: sundials4py.core._generic_N_Vector, y: sundials4py.core._generic_N_Vector) → float`  
See [`N\_VDotProd\(\)`](#).

`sundials4py.core.N_VDotProdLocal(x: sundials4py.core._generic_N_Vector, y: sundials4py.core._generic_N_Vector) → float`  
See [`N\_VDotProdLocal\(\)`](#).

`sundials4py.core.N_VDotProdMulti(nvec: int, x: sundials4py.core._generic_N_Vector, Y_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], dotprods_1d: numpy.ndarray[dtype=float64, shape=(*), order='C']) → int`  
See [`N\_VDotProdMulti\(\)`](#).

`sundials4py.core.N_VDotProdMultiAllReduce(nvec_total: int, x: sundials4py.core._generic_N_Vector, sum_1d: numpy.ndarray[dtype=float64, shape=(*), order='C']) → int`  
See [`N\_VDotProdMultiAllReduce\(\)`](#).

`sundials4py.core.N_VDotProdMultiLocal(nvec: int, x: sundials4py.core._generic_N_Vector, Y_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], dotprods_1d: numpy.ndarray[dtype=float64, shape=(*), order='C']) → int`



See `N_VDotProdMultiLocal()`.

`sundials4py.core.N_VEnableConstVectorArray_ManyVector(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableConstVectorArray_ManyVector()`.

`sundials4py.core.N_VEnableConstVectorArray_Serial(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableConstVectorArray_Serial()`.

`sundials4py.core.N_VEnableDotProdMultiLocal_ManyVector(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableDotProdMultiLocal_ManyVector()`.

`sundials4py.core.N_VEnableDotProdMulti_ManyVector(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableDotProdMulti_ManyVector()`.

`sundials4py.core.N_VEnableDotProdMulti_Serial(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableDotProdMulti_Serial()`.

`sundials4py.core.N_VEnableFusedOps_ManyVector(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableFusedOps_ManyVector()`.

`sundials4py.core.N_VEnableFusedOps_Serial(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableFusedOps_Serial()`.

`sundials4py.core.N_VEnableLinearCombinationVectorArray_Serial(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableLinearCombinationVectorArray_Serial()`.

`sundials4py.core.N_VEnableLinearCombination_ManyVector(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableLinearCombination_ManyVector()`.

`sundials4py.core.N_VEnableLinearCombination_Serial(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableLinearCombination_Serial()`.

`sundials4py.core.N_VEnableLinearSumVectorArray_ManyVector(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableLinearSumVectorArray_ManyVector()`.

`sundials4py.core.N_VEnableLinearSumVectorArray_Serial(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableLinearSumVectorArray_Serial()`.

`sundials4py.core.N_VEnableScaleAddMultiVectorArray_Serial(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableScaleAddMultiVectorArray_Serial()`.

`sundials4py.core.N_VEnableScaleAddMulti_ManyVector(v: sundials4py.core._generic_N_Vector, tf: int) → int`

See `N_VEnableScaleAddMulti_ManyVector()`.

`sundials4py.core.N_VEnableScaleAddMulti_Serial`(*v*: `sundials4py.core._generic_N_Vector`, *tf*: *int*) → *int*  
See `N_VEnableScaleAddMulti_Serial()`.

`sundials4py.core.N_VEnableScaleVectorArray_ManyVector`(*v*: `sundials4py.core._generic_N_Vector`, *tf*: *int*) → *int*  
See `N_VEnableScaleVectorArray_ManyVector()`.

`sundials4py.core.N_VEnableScaleVectorArray_Serial`(*v*: `sundials4py.core._generic_N_Vector`, *tf*: *int*) → *int*  
See `N_VEnableScaleVectorArray_Serial()`.

`sundials4py.core.N_VEnableWrmsNormMaskVectorArray_ManyVector`(*v*: `sundials4py.core._generic_N_Vector`, *tf*: *int*) → *int*  
See `N_VEnableWrmsNormMaskVectorArray_ManyVector()`.

`sundials4py.core.N_VEnableWrmsNormMaskVectorArray_Serial`(*v*: `sundials4py.core._generic_N_Vector`, *tf*: *int*) → *int*  
See `N_VEnableWrmsNormMaskVectorArray_Serial()`.

`sundials4py.core.N_VEnableWrmsNormVectorArray_ManyVector`(*v*: `sundials4py.core._generic_N_Vector`, *tf*: *int*) → *int*  
See `N_VEnableWrmsNormVectorArray_ManyVector()`.

`sundials4py.core.N_VEnableWrmsNormVectorArray_Serial`(*v*: `sundials4py.core._generic_N_Vector`, *tf*: *int*) → *int*  
See `N_VEnableWrmsNormVectorArray_Serial()`.

`sundials4py.core.N_VGetArrayPointer`(*arg*: `sundials4py.core._generic_N_Vector`, *l* (*Positional-only parameter separator (PEP 570)*)) → `numpy.ndarray`[*dtype*=float64, *shape*=(\*), *order*='C']  
See `N_VGetArrayPointer()`.

`sundials4py.core.N_VGetCommunicator`(*v*: `sundials4py.core._generic_N_Vector`) → *int*  
See `N_VGetCommunicator()`.

`sundials4py.core.N_VGetDeviceArrayPointer`(*arg*: `sundials4py.core._generic_N_Vector`, *l*) → `numpy.ndarray`[*dtype*=float64, *shape*=(\*), *order*='C']  
See `N_VGetDeviceArrayPointer()`.

`sundials4py.core.N_VGetLength`(*v*: `sundials4py.core._generic_N_Vector`) → *int*  
See `N_VGetLength()`.

`sundials4py.core.N_VGetLocalLength`(*v*: `sundials4py.core._generic_N_Vector`) → *int*  
See `N_VGetLocalLength()`.

`sundials4py.core.N_VGetNumSubvectors_ManyVector`(*v*: `sundials4py.core._generic_N_Vector`) → *int*  
See `N_VGetNumSubvectors_ManyVector()`.

`sundials4py.core.N_VGetSubvectorLocalLength_ManyVector`(*v*: `sundials4py.core._generic_N_Vector`, *vec\_num*: *int*) → *int*  
See `N_VGetSubvectorLocalLength_ManyVector()`.

`sundials4py.core.N_VGetSubvector_ManyVector`(*v*: `sundials4py.core._generic_N_Vector`, *vec\_num*: *int*) → `sundials4py.core._generic_N_Vector`  
*nb::rv\_policy::reference*  
See `N_VGetSubvector_ManyVector()`.

`sundials4py.core.N_VGetVectorID(w: sundials4py.core._generic_N_Vector) →`  
*sundials4py.core.N\_Vector\_ID*

See *N\_VGetVectorID()*.

`sundials4py.core.N_VInv(x: sundials4py.core._generic_N_Vector, z: sundials4py.core._generic_N_Vector) →`  
 None

See *N\_VInv()*.

`sundials4py.core.N_VInvTest(x: sundials4py.core._generic_N_Vector, z:`  
*sundials4py.core.\_generic\_N\_Vector) → int*

See *N\_VInvTest()*.

`sundials4py.core.N_VInvTestLocal(x: sundials4py.core._generic_N_Vector, z:`  
*sundials4py.core.\_generic\_N\_Vector) → int*

See *N\_VInvTestLocal()*.

`sundials4py.core.N_VL1Norm(x: sundials4py.core._generic_N_Vector) → float`

See *N\_VL1Norm()*.

`sundials4py.core.N_VL1NormLocal(x: sundials4py.core._generic_N_Vector) → float`

See *N\_VL1NormLocal()*.

`sundials4py.core.N_VLinearCombination(nvec: int, c_1d: numpy.ndarray[dtype=float64, shape=(*)],`  
*order='C'], X\_1d:*  
*collections.abc.Sequence[sundials4py.core.\_generic\_N\_Vector], z:*  
*sundials4py.core.\_generic\_N\_Vector) → int*

See *N\_VLinearCombination()*.

`sundials4py.core.N_VLinearCombinationVectorArray(nvec: int, nsum: int, c_1d:`  
*numpy.ndarray[dtype=float64, shape=(\*)],*  
*order='C'], X\_2d: collec-*  
*tions.abc.Sequence[collections.abc.Sequence[sundials4py.core.\_*  
*generic\_N\_Vector]], Z\_1d:*  
*collections.abc.Sequence[sundials4py.core.\_*  
*generic\_N\_Vector]) →*  
 int

See *N\_VLinearCombinationVectorArray()*.

`sundials4py.core.N_VLinearSum(a: float, x: sundials4py.core._generic_N_Vector, b: float, y:`  
*sundials4py.core.\_generic\_N\_Vector, z:*  
*sundials4py.core.\_generic\_N\_Vector) → None*

See *N\_VLinearSum()*.

`sundials4py.core.N_VLinearSumVectorArray(nvec: int, a: float, X_1d:`  
*collections.abc.Sequence[sundials4py.core.\_generic\_N\_*  
*Vector], b: float, Y\_1d:*  
*collections.abc.Sequence[sundials4py.core.\_generic\_N\_*  
*Vector], Z\_1d:*  
*collections.abc.Sequence[sundials4py.core.\_generic\_N\_*  
*Vector]) →*  
 int

See *N\_VLinearSumVectorArray()*.

`sundials4py.core.N_VMake_Serial(vec_length: int, v_data_1d: numpy.ndarray[dtype=float64, shape=(*)],`  
*order='C'], sunctx: sundials4py.core.SUNContext\_)* →  
*sundials4py.core.\_generic\_N\_Vector*

`nb::keep_alive<0, 3>()`

See `N_VMake_Serial()`.

`sundials4py.core.N_VMaxNorm(x: sundials4py.core._generic_N_Vector) → float`

See `N_VMaxNorm()`.

`sundials4py.core.N_VMaxNormLocal(x: sundials4py.core._generic_N_Vector) → float`

See `N_VMaxNormLocal()`.

`sundials4py.core.N_VMin(x: sundials4py.core._generic_N_Vector) → float`

See `N_VMin()`.

`sundials4py.core.N_VMinLocal(x: sundials4py.core._generic_N_Vector) → float`

See `N_VMinLocal()`.

`sundials4py.core.N_VMinQuotient(num: sundials4py.core._generic_N_Vector, denom:  
sundials4py.core._generic_N_Vector) → float`

See `N_VMinQuotient()`.

`sundials4py.core.N_VMinQuotientLocal(num: sundials4py.core._generic_N_Vector, denom:  
sundials4py.core._generic_N_Vector) → float`

See `N_VMinQuotientLocal()`.

`sundials4py.core.N_VNewEmpty_Serial(vec_length: int, sunctx: sundials4py.core.SUNContext_) →  
sundials4py.core._generic_N_Vector`

`nb::keep_alive<0, 2>()`

See `N_VNewEmpty_Serial()`.

`sundials4py.core.N_VNew_ManyVector(num_subvectors: int, vec_array_id:  
collections.abc.Sequence[sundials4py.core._generic_N_Vector],  
sunctx: sundials4py.core.SUNContext_) →  
sundials4py.core._generic_N_Vector`

See `N_VNew_ManyVector()`.

`sundials4py.core.N_VNew_Serial(vec_length: int, sunctx: sundials4py.core.SUNContext_) →  
sundials4py.core._generic_N_Vector`

`nb::keep_alive<0, 2>()`

See `N_VNew_Serial()`.

`sundials4py.core.N_VPrint(v: sundials4py.core._generic_N_Vector) → None`

See `N_VPrint()`.

`sundials4py.core.N_VPrintFile(v: sundials4py.core._generic_N_Vector, outfile: sundials4py.core.FILE) →  
None`

See `N_VPrintFile()`.

`sundials4py.core.N_VProd(x: sundials4py.core._generic_N_Vector, y: sundials4py.core._generic_N_Vector, z:  
sundials4py.core._generic_N_Vector) → None`

See `N_VProd()`.

`sundials4py.core.N_VScale(c: float, x: sundials4py.core._generic_N_Vector, z:  
sundials4py.core._generic_N_Vector) → None`

See `N_VScale()`.

`sundials4py.core.N_VScaleAddMulti`(*nvec*: int, *a\_1d*: numpy.ndarray[dtype=float64, shape=(\*), order='C'],  
*x*: sundials4py.core.\_generic\_N\_Vector, *Y\_1d*:  
collections.abc.Sequence[sundials4py.core.\_generic\_N\_Vector], *Z\_1d*:  
collections.abc.Sequence[sundials4py.core.\_generic\_N\_Vector]) → int

See `N_VScaleAddMulti()`.

`sundials4py.core.N_VScaleAddMultiVectorArray`(*nvec*: int, *nsum*: int, *c\_1d*: numpy.ndarray[dtype=float64,  
shape=(\*), order='C'], *X\_1d*:  
collections.abc.Sequence[sundials4py.core.\_generic\_N\_  
Vector], *Y\_2d*:  
collec-  
tions.abc.Sequence[collections.abc.Sequence[sundials4py.core.\_  
generic\_N\_Vector]], *Z\_2d*:  
collec-  
tions.abc.Sequence[collections.abc.Sequence[sundials4py.core.\_  
generic\_N\_Vector]]) →  
int

See `N_VScaleAddMultiVectorArray()`.

`sundials4py.core.N_VScaleVectorArray`(*nvec*: int, *c\_1d*: numpy.ndarray[dtype=float64, shape=(\*),  
order='C'], *X\_1d*:  
collections.abc.Sequence[sundials4py.core.\_generic\_N\_Vector],  
*Z\_1d*:  
collections.abc.Sequence[sundials4py.core.\_generic\_N\_Vector]) →  
int

See `N_VScaleVectorArray()`.

`sundials4py.core.N_VSetArrayPointer`(*arg0*: numpy.ndarray[dtype=float64, shape=(\*), order='C'], *arg1*:  
sundials4py.core.\_generic\_N\_Vector, /) → None

See `N_VSetArrayPointer()`.

`sundials4py.core.N_VWL2Norm`(*x*: sundials4py.core.\_generic\_N\_Vector, *w*:  
sundials4py.core.\_generic\_N\_Vector) → float

See `N_VWL2Norm()`.

`sundials4py.core.N_VWSqrSumLocal`(*x*: sundials4py.core.\_generic\_N\_Vector, *w*:  
sundials4py.core.\_generic\_N\_Vector) → float

See `N_VWSqrSumLocal()`.

`sundials4py.core.N_VWSqrSumMaskLocal`(*x*: sundials4py.core.\_generic\_N\_Vector, *w*:  
sundials4py.core.\_generic\_N\_Vector, *id*:  
sundials4py.core.\_generic\_N\_Vector) → float

See `N_VWSqrSumMaskLocal()`.

`sundials4py.core.N_VWrmsNorm`(*x*: sundials4py.core.\_generic\_N\_Vector, *w*:  
sundials4py.core.\_generic\_N\_Vector) → float

See `N_VWrmsNorm()`.

`sundials4py.core.N_VWrmsNormMask`(*x*: sundials4py.core.\_generic\_N\_Vector, *w*:  
sundials4py.core.\_generic\_N\_Vector, *id*:  
sundials4py.core.\_generic\_N\_Vector) → float

See `N_VWrmsNormMask()`.

`sundials4py.core.N_VWrmsNormMaskVectorArray`(*nvec*: int, *X\_1d*:  
*collections.abc.Sequence*[*sundials4py.core.\_generic\_N\_*  
*Vector*], *W\_1d*:  
*collections.abc.Sequence*[*sundials4py.core.\_generic\_N\_*  
*Vector*], *id*: *sundials4py.core.\_generic\_N\_Vector*; *nrm\_1d*:  
*numpy.ndarray*[*dtype=float64*, *shape*=(\*)], *order*='C']) →  
int

See [`N\_VWrmsNormMaskVectorArray\(\)`](#).

`sundials4py.core.N_VWrmsNormVectorArray`(*nvec*: int, *X\_1d*:  
*collections.abc.Sequence*[*sundials4py.core.\_generic\_N\_Vector*],  
*W\_1d*:  
*collections.abc.Sequence*[*sundials4py.core.\_generic\_N\_Vector*],  
*nrm\_1d*: *numpy.ndarray*[*dtype=float64*, *shape*=(\*)], *order*='C'])  
→ int

See [`N\_VWrmsNormVectorArray\(\)`](#).

`sundials4py.core.SUNAbortErrHandlerFn`(*line*: int, *func*: str, *file*: str, *msg*: str, *err\_code*: int, *err\_user\_data*:  
*typing\_extensions.CapsuleType*, *sunctx*:  
*sundials4py.core.SUNContext\_*) → None

See [`SUNAbortErrHandlerFn\(\)`](#).

`sundials4py.core.SUNAdaptController_EstimateStep`(*C*: *sundials4py.core.\_generic\_SUNAdaptController*,  
*h*: float, *p*: int, *dsm*: float) → tuple[int, float]

See [`SUNAdaptController\_EstimateStep\(\)`](#).

`sundials4py.core.SUNAdaptController_EstimateStepTol`(*C*:  
*sundials4py.core.\_generic\_SUNAdaptController*,  
*H*: float, *tolfac*: float, *P*: int, *DSM*: float, *dsm*:  
float) → tuple[int, float, float]

See [`SUNAdaptController\_EstimateStepTol\(\)`](#).

`sundials4py.core.SUNAdaptController_ExpGus`(*sunctx*: *sundials4py.core.SUNContext\_*) →  
*sundials4py.core.\_generic\_SUNAdaptController*

*nb::keep\_alive*<0, 1>()

See [`SUNAdaptController\_ExpGus\(\)`](#).

`sundials4py.core.SUNAdaptController_GetFastController_MRIHTol`(*C*: *sundials4py.core.\_generic\_*  
*SUNAdaptController*) → tuple[int,  
*sundials4py.core.\_generic\_*  
*SUNAdaptController*]

*nb::rv\_policy::reference*

See [`SUNAdaptController\_GetFastController\_MRIHTol\(\)`](#).

`sundials4py.core.SUNAdaptController_GetSlowController_MRIHTol`(*C*: *sundials4py.core.\_generic\_*  
*SUNAdaptController*) → tuple[int,  
*sundials4py.core.\_generic\_*  
*SUNAdaptController*]

*nb::rv\_policy::reference*

See [`SUNAdaptController\_GetSlowController\_MRIHTol\(\)`](#).

`sundials4py.core.SUNAdaptController_GetType`(*C*: *sundials4py.core.\_generic\_SUNAdaptController*) →  
*sundials4py.core.SUNAdaptController\_Type*

See [`SUNAdaptController\_GetType\(\)`](#).

`sundials4py.core.SUNAdaptController_H0211(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_H0211()*.

`sundials4py.core.SUNAdaptController_H0321(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_H0321()*.

`sundials4py.core.SUNAdaptController_H211(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_H211()*.

`sundials4py.core.SUNAdaptController_H312(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_H312()*.

`sundials4py.core.SUNAdaptController_I(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_I()*.

`sundials4py.core.SUNAdaptController_ImExGus(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_ImExGus()*.

`sundials4py.core.SUNAdaptController_ImpGus(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_ImpGus()*.

`sundials4py.core.SUNAdaptController_MRIHTol(HControl:`  
*sundials4py.core.\_generic\_SUNAdaptController,*  
*TolControl:*  
*sundials4py.core.\_generic\_SUNAdaptController, sunctx:*  
*sundials4py.core.SUNContext\_) →*  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 3>()`

See *SUNAdaptController\_MRIHTol()*.

`sundials4py.core.SUNAdaptController_PI(sunctx: sundials4py.core.SUNContext_) →`  
*sundials4py.core.\_generic\_SUNAdaptController*

`nb::keep_alive<0, 1>()`

See *SUNAdaptController\_PI()*.



`sundials4py.core.SUNAdaptController_PID`(*sunctx*: `sundials4py.core.SUNContext_`) → `sundials4py.core._generic_SUNAdaptController`

`nb::keep_alive<0, 1>()`

See `SUNAdaptController_PID()`.

`sundials4py.core.SUNAdaptController_Reset`(*C*: `sundials4py.core._generic_SUNAdaptController`) → `int`

See `SUNAdaptController_Reset()`.

`sundials4py.core.SUNAdaptController_SetDefaults`(*C*: `sundials4py.core._generic_SUNAdaptController`) → `int`

See `SUNAdaptController_SetDefaults()`.

`sundials4py.core.SUNAdaptController_SetErrorBias`(*C*: `sundials4py.core._generic_SUNAdaptController`, *bias*: `float`) → `int`

See `SUNAdaptController_SetErrorBias()`.

`sundials4py.core.SUNAdaptController_SetOptions`(*self*: `sundials4py.core._generic_SUNAdaptController`, *id*: `str`, *file\_name*: `str`, *argc*: `int`, *args*: `collections.abc.Sequence[str]`) → `int`

See `SUNAdaptController_SetOptions()`.

`sundials4py.core.SUNAdaptController_SetParams_ExpGus`(*C*: `sundials4py.core._generic_SUNAdaptController`, *k1*: `float`, *k2*: `float`) → `int`

See `SUNAdaptController_SetParams_ExpGus()`.

`sundials4py.core.SUNAdaptController_SetParams_I`(*C*: `sundials4py.core._generic_SUNAdaptController`, *k1*: `float`) → `int`

See `SUNAdaptController_SetParams_I()`.

`sundials4py.core.SUNAdaptController_SetParams_ImExGus`(*C*: `sundials4py.core._generic_SUNAdaptController`, *k1e*: `float`, *k2e*: `float`, *k1i*: `float`, *k2i*: `float`) → `int`

See `SUNAdaptController_SetParams_ImExGus()`.

`sundials4py.core.SUNAdaptController_SetParams_ImpGus`(*C*: `sundials4py.core._generic_SUNAdaptController`, *k1*: `float`, *k2*: `float`) → `int`

See `SUNAdaptController_SetParams_ImpGus()`.

`sundials4py.core.SUNAdaptController_SetParams_MRIHTol`(*C*: `sundials4py.core._generic_SUNAdaptController`, *inner\_max\_relch*: `float`, *inner\_min\_tolfac*: `float`, *inner\_max\_tolfac*: `float`) → `int`

See `SUNAdaptController_SetParams_MRIHTol()`.

`sundials4py.core.SUNAdaptController_SetParams_PI`(*C*: `sundials4py.core._generic_SUNAdaptController`, *k1*: `float`, *k2*: `float`) → `int`

See `SUNAdaptController_SetParams_PI()`.

`sundials4py.core.SUNAdaptController_SetParams_PID`(*C*: `sundials4py.core._generic_SUNAdaptController`, *k1*: `float`, *k2*: `float`, *k3*: `float`) → `int`

See `SUNAdaptController_SetParams_PID()`.



`sundials4py.core.SUNAdaptController_SetParams_Soderlind`(*C*: `sundials4py.core._generic_SUNAdaptController`, *k1*: `float`, *k2*: `float`, *k3*: `float`, *k4*: `float`, *k5*: `float`) → `int`

See `SUNAdaptController_SetParams_Soderlind()`.

`sundials4py.core.SUNAdaptController_Soderlind`(*sunctx*: `sundials4py.core.SUNContext_`) → `sundials4py.core._generic_SUNAdaptController`

*nb::keep\_alive*<0, 1>()

See `SUNAdaptController_Soderlind()`.

`sundials4py.core.SUNAdaptController_UpdateH`(*C*: `sundials4py.core._generic_SUNAdaptController`, *h*: `float`, *dsm*: `float`) → `int`

See `SUNAdaptController_UpdateH()`.

`sundials4py.core.SUNAdaptController_UpdateMRIHTol`(*C*: `sundials4py.core._generic_SUNAdaptController`, *H*: `float`, *tolfac*: `float`, *DSM*: `float`, *dsm*: `float`) → `int`

See `SUNAdaptController_UpdateMRIHTol()`.

`sundials4py.core.SUNAdaptController_Write`(*C*: `sundials4py.core._generic_SUNAdaptController`, *fptr*: `sundials4py.core.FILE`) → `int`

See `SUNAdaptController_Write()`.

`sundials4py.core.SUNAdjointCheckpointScheme_Create_Fixed`(*io\_mode*: `sundials4py.core.SUNDataIOMode`, *mem\_helper*: `sundials4py.core.SUNMemoryHelper_`, *interval*: `int`, *estimate*: `int`, *keep*: `int`, *sunctx*: `sundials4py.core.SUNContext_`) → `tuple[int, sundials4py.core.SUNAdjointCheckpointScheme_]`

See `SUNAdjointCheckpointScheme_Create_Fixed()`.

`sundials4py.core.SUNAdjointCheckpointScheme_EnableDense`(*check\_scheme*: `sundials4py.core.SUNAdjointCheckpointScheme_`, *on\_or\_off*: `int`) → `int`

See `SUNAdjointCheckpointScheme_EnableDense()`.

`sundials4py.core.SUNAdjointCheckpointScheme_InsertVector`(*check\_scheme*: `sundials4py.core.SUNAdjointCheckpointScheme_`, *step\_num*: `int`, *stage\_num*: `int`, *t*: `float`, *state*: `sundials4py.core._generic_N_Vector`) → `int`

See `SUNAdjointCheckpointScheme_InsertVector()`.

`sundials4py.core.SUNAdjointCheckpointScheme_LoadVector`(*check\_scheme*: `sundials4py.core.SUNAdjointCheckpointScheme_`, *step\_num*: `int`, *stage\_num*: `int`, *peek*: `int`, *tmpl*: `sundials4py.core._generic_N_Vector`) → `tuple[int, sundials4py.core._generic_N_Vector, float]`

See *SUNAdjointCheckpointScheme\_LoadVector()*.

`sundials4py.core.SUNAdjointCheckpointScheme_NeedsSaving`(*check\_scheme*: `sundials4py.core.SUNAdjointCheckpointScheme_`, *step\_num*: `int`, *stage\_num*: `int`, *t*: `float`) → `tuple[int, int]`

See *SUNAdjointCheckpointScheme\_NeedsSaving()*.

`sundials4py.core.SUNAdjointStepper_Create`(*fwd\_sunstepper*: `sundials4py.core.SUNStepper_`, *own\_fwd*: `int`, *adj\_sunstepper*: `sundials4py.core.SUNStepper_`, *own\_adj*: `int`, *final\_step\_idx*: `int`, *tf*: `float`, *sf*: `sundials4py.core._generic_N_Vector`, *checkpoint\_scheme*: `sundials4py.core.SUNAdjointCheckpointScheme_`, *sunctx*: `sundials4py.core.SUNContext_`) → `tuple[int, sundials4py.core.SUNAdjointStepper_]`

`nb::call_policy<sundials4py::returns_references_to<9, 1>>()`

See *SUNAdjointStepper\_Create()*.

`sundials4py.core.SUNAdjointStepper_Evolve`(*adj\_stepper*: `sundials4py.core.SUNAdjointStepper_`, *tout*: `float`, *sens*: `sundials4py.core._generic_N_Vector`) → `tuple[int, float]`

See *SUNAdjointStepper\_Evolve()*.

`sundials4py.core.SUNAdjointStepper_GetNumRecompute`(*adj\_stepper*: `sundials4py.core.SUNAdjointStepper_`) → `tuple[int, int]`

See *SUNAdjointStepper\_GetNumRecompute()*.

`sundials4py.core.SUNAdjointStepper_GetNumSteps`(*adj\_stepper*: `sundials4py.core.SUNAdjointStepper_`) → `tuple[int, int]`

See *SUNAdjointStepper\_GetNumSteps()*.

`sundials4py.core.SUNAdjointStepper_OneStep`(*adj\_stepper*: `sundials4py.core.SUNAdjointStepper_`, *tout*: `float`, *sens*: `sundials4py.core._generic_N_Vector`) → `tuple[int, float]`

See *SUNAdjointStepper\_OneStep()*.

`sundials4py.core.SUNAdjointStepper_PrintAllStats`(*adj\_stepper*: `sundials4py.core.SUNAdjointStepper_`, *outfile*: `sundials4py.core.FILE`, *fmt*: `sundials4py.core.SUNOutputFormat`) → `int`

See *SUNAdjointStepper\_PrintAllStats()*.

`sundials4py.core.SUNAdjointStepper_ReInit`(*adj*: `sundials4py.core.SUNAdjointStepper_`, *t0*: `float`, *y0*: `sundials4py.core._generic_N_Vector`, *tf*: `float`, *sf*: `sundials4py.core._generic_N_Vector`) → `int`

See *SUNAdjointStepper\_ReInit()*.

`sundials4py.core.SUNAdjointStepper_RecomputeFwd`(*adj\_stepper*: `sundials4py.core.SUNAdjointStepper_`, *start\_idx*: `int`, *t0*: `float`, *y0*: `sundials4py.core._generic_N_Vector`, *tf*: `float`) → `int`

See *SUNAdjointStepper\_RecomputeFwd()*.

`sundials4py.core.SUNAdjointStepper_SetUserData`(*param\_0*: `sundials4py.core.SUNAdjointStepper_`, *user\_data*: `typing_extensions.CapsuleType`) → `int`

See *SUNAdjointStepper\_SetUserData()*.

`sundials4py.core.SUNBandMatrix`(*N*: int, *mu*: int, *ml*: int, *sunctx*: `sundials4py.core.SUNContext_`) → `sundials4py.core._generic_SUNMatrix`

`nb::keep_alive<0, 4>()`

See `SUNBandMatrix()`.

`sundials4py.core.SUNBandMatrixStorage`(*N*: int, *mu*: int, *ml*: int, *smu*: int, *sunctx*: `sundials4py.core.SUNContext_`) → `sundials4py.core._generic_SUNMatrix`

`nb::keep_alive<0, 5>()`

See `SUNBandMatrixStorage()`.

`sundials4py.core.SUNBandMatrix_Columns`(*A*: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNBandMatrix_Columns()`.

`sundials4py.core.SUNBandMatrix_Data`(*A*: `sundials4py.core._generic_SUNMatrix`) → `numpy.ndarray`[`dtype=float64`, `shape=()`, `order='C'`]

See `SUNBandMatrix_Data()`.

`sundials4py.core.SUNBandMatrix_LData`(*A*: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNBandMatrix_LData()`.

`sundials4py.core.SUNBandMatrix_LDim`(*A*: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNBandMatrix_LDim()`.

`sundials4py.core.SUNBandMatrix_LowerBandwidth`(*A*: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNBandMatrix_LowerBandwidth()`.

`sundials4py.core.SUNBandMatrix_Print`(*A*: `sundials4py.core._generic_SUNMatrix`, *outfile*: `sundials4py.core.FILE`) → None

See `SUNBandMatrix_Print()`.

`sundials4py.core.SUNBandMatrix_Rows`(*A*: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNBandMatrix_Rows()`.

`sundials4py.core.SUNBandMatrix_StoredUpperBandwidth`(*A*: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNBandMatrix_StoredUpperBandwidth()`.

`sundials4py.core.SUNBandMatrix_UpperBandwidth`(*A*: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNBandMatrix_UpperBandwidth()`.

`sundials4py.core.SUNClassicalGS`(*v\_1d*: `collections.abc.Sequence`[`sundials4py.core._generic_N_Vector`], *h\_2d*: `numpy.ndarray`[`dtype=float64`, `shape=()`, `order='C'`], *k*: int, *p*: int, *stemp\_1d*: `numpy.ndarray`[`dtype=float64`, `shape=()`, `order='C'`], *vtemp\_1d*: `collections.abc.Sequence`[`sundials4py.core._generic_N_Vector`]) → `tuple`[int, float]

See `SUNClassicalGS()`.

`sundials4py.core.SUNContext_ClearErrHandlers`(*sunctx*: `sundials4py.core.SUNContext_`) → int

See `SUNContext_ClearErrHandlers()`.

`sundials4py.core.SUNContext_Create`(*comm*: int) → `tuple`[int, `sundials4py.core.SUNContext_`]

See `SUNContext_Create()`.

`sundials4py.core.SUNContext_GetLastError(sunctx: sundials4py.core.SUNContext_) → int`  
See `SUNContext_GetLastError()`.

`sundials4py.core.SUNContext_GetLogger(sunctx: sundials4py.core.SUNContext_) → tuple[int, sundials4py.core.SUNLogger_]`  
`nb::rv_policy::reference`  
See `SUNContext_GetLogger()`.

`sundials4py.core.SUNContext_GetProfiler(sunctx: sundials4py.core.SUNContext_) → tuple[int, sundials4py.core.SUNProfiler_]`  
`nb::rv_policy::reference`  
See `SUNContext_GetProfiler()`.

`sundials4py.core.SUNContext_PeekLastError(sunctx: sundials4py.core.SUNContext_) → int`  
See `SUNContext_PeekLastError()`.

`sundials4py.core.SUNContext_PopErrorHandler(arg: sundials4py.core.SUNContext_, /) → int`  
See `SUNContext_PopErrorHandler()`.

`sundials4py.core.SUNContext_PushErrorHandler(arg0: sundials4py.core.SUNContext_, arg1: collections.abc.Callable[[int, str, str, str, int, typing_extensions.CapsuleType, sundials4py.core.SUNContext_], None], /) → sundials4py.core.SUNErrCode`  
See `SUNContext_PushErrorHandler()`.

`sundials4py.core.SUNContext_SetLogger(sunctx: sundials4py.core.SUNContext_, logger: sundials4py.core.SUNLogger_) → int`  
See `SUNContext_SetLogger()`.

`sundials4py.core.SUNContext_SetProfiler(sunctx: sundials4py.core.SUNContext_, profiler: sundials4py.core.SUNProfiler_) → int`  
See `SUNContext_SetProfiler()`.

`sundials4py.core.SUNDenseMatrix(M: int, N: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNMatrix`  
`nb::keep_alive<0, 3>()`  
See `SUNDenseMatrix()`.

`sundials4py.core.SUNDenseMatrix_Columns(A: sundials4py.core._generic_SUNMatrix) → int`  
See `SUNDenseMatrix_Columns()`.

`sundials4py.core.SUNDenseMatrix_Data(A: sundials4py.core._generic_SUNMatrix) → numpy.ndarray[dtype=float64, shape=(*, *), order='F']`  
See `SUNDenseMatrix_Data()`.

`sundials4py.core.SUNDenseMatrix_LData(A: sundials4py.core._generic_SUNMatrix) → int`  
See `SUNDenseMatrix_LData()`.

`sundials4py.core.SUNDenseMatrix_Print(A: sundials4py.core._generic_SUNMatrix, outfile: sundials4py.core.FILE) → None`  
See `SUNDenseMatrix_Print()`.

`sundials4py.core.SUNDenseMatrix_Rows(A: sundials4py.core._generic_SUNMatrix) → int`  
See `SUNDenseMatrix_Rows()`.

`sundials4py.core.SUNDomEigEstimator_Estimate`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`) →  
tuple[int, float, float]

See `SUNDomEigEstimator_Estimate()`.

`sundials4py.core.SUNDomEigEstimator_GetNumATimesCalls`(*DEE*:  
`sundials4py.core.SUNDomEigEstimator_`) →  
tuple[int, int]

See `SUNDomEigEstimator_GetNumATimesCalls()`.

`sundials4py.core.SUNDomEigEstimator_GetNumIters`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`) →  
tuple[int, int]

See `SUNDomEigEstimator_GetNumIters()`.

`sundials4py.core.SUNDomEigEstimator_GetRes`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`) →  
tuple[int, float]

See `SUNDomEigEstimator_GetRes()`.

`sundials4py.core.SUNDomEigEstimator_Initialize`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`) → int  
See `SUNDomEigEstimator_Initialize()`.

`sundials4py.core.SUNDomEigEstimator_Power`(*q*: `sundials4py.core._generic_N_Vector`, *max\_iters*: int,  
*rel\_tol*: float, *sunctx*: `sundials4py.core.SUNContext_`) →  
`sundials4py.core.SUNDomEigEstimator_`

`nb::keep_alive<0, 4>()`

See `SUNDomEigEstimator_Power()`.

`sundials4py.core.SUNDomEigEstimator_SetATimes`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`,  
*ATimes*: `collections.abc.Callable[[typing_ -  
extensions.CapsuleType,  
sundials4py.core._generic_N_Vector,  
sundials4py.core._generic_N_Vector], int] | None`) → int

See `SUNDomEigEstimator_SetATimes()`.

`sundials4py.core.SUNDomEigEstimator_SetInitialGuess`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`,  
*q*: `sundials4py.core._generic_N_Vector`) → int

See `SUNDomEigEstimator_SetInitialGuess()`.

`sundials4py.core.SUNDomEigEstimator_SetMaxIters`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`,  
*max\_iters*: int) → int

See `SUNDomEigEstimator_SetMaxIters()`.

`sundials4py.core.SUNDomEigEstimator_SetNumPreprocessIters`(*DEE*: `sundi -  
als4py.core.SUNDomEigEstimator_`,  
*num\_iters*: int) → int

See `SUNDomEigEstimator_SetNumPreprocessIters()`.

`sundials4py.core.SUNDomEigEstimator_SetOptions`(*self*: `sundials4py.core.SUNDomEigEstimator_`, *id*: str,  
*file\_name*: str, *argc*: int, *args*:  
`collections.abc.Sequence[str]`) → int

See `SUNDomEigEstimator_SetOptions()`.

`sundials4py.core.SUNDomEigEstimator_SetRelTol`(*DEE*: `sundials4py.core.SUNDomEigEstimator_`, *tol*:  
float) → int

See `SUNDomEigEstimator_SetRelTol()`.

`sundials4py.core.SUNDomEigEstimator_Write(DEE: sundials4py.core.SUNDomEigEstimator_, outfile: sundials4py.core.FILE) → int`

See `SUNDomEigEstimator_Write()`.

`sundials4py.core.SUNFileOpen(arg0: str, arg1: str, /) → tuple[int, sundials4py.core.FILE]`

See `SUNFileOpen()`.

`sundials4py.core.SUNGetErrMsg(code: int) → str`

See `SUNGetErrMsg()`.

`sundials4py.core.SUNLinSolGetID(S: sundials4py.core._generic_SUNLinearSolver) → sundials4py.core.SUNLinearSolver_ID`

See `SUNLinSolGetID()`.

`sundials4py.core.SUNLinSolGetType(S: sundials4py.core._generic_SUNLinearSolver) → sundials4py.core.SUNLinearSolver_Type`

See `SUNLinSolGetType()`.

`sundials4py.core.SUNLinSolInitialize(S: sundials4py.core._generic_SUNLinearSolver) → int`

See `SUNLinSolInitialize()`.

`sundials4py.core.SUNLinSolLastFlag(S: sundials4py.core._generic_SUNLinearSolver) → int`

See `SUNLinSolLastFlag()`.

`sundials4py.core.SUNLinSolNumIters(S: sundials4py.core._generic_SUNLinearSolver) → int`

See `SUNLinSolNumIters()`.

`sundials4py.core.SUNLinSolResNorm(S: sundials4py.core._generic_SUNLinearSolver) → float`

See `SUNLinSolResNorm()`.

`sundials4py.core.SUNLinSolResid(S: sundials4py.core._generic_SUNLinearSolver) → sundials4py.core._generic_N_Vector`

nb::rv\_policy::reference

See `SUNLinSolResid()`.

`sundials4py.core.SUNLinSolSetATimes(LS: sundials4py.core._generic_SUNLinearSolver, ATimes: collections.abc.Callable[[typing_extensions.CapsuleType, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector], int] | None) → int`

See `SUNLinSolSetATimes()`.

`sundials4py.core.SUNLinSolSetOptions(self: sundials4py.core._generic_SUNLinearSolver, id: str, file_name: str, argc: int, args: collections.abc.Sequence[str]) → int`

See `SUNLinSolSetOptions()`.

`sundials4py.core.SUNLinSolSetPreconditioner(LS: sundials4py.core._generic_SUNLinearSolver, PSetupFn: collections.abc.Callable[[typing_extensions.CapsuleType], int] | None, PSolveFn: collections.abc.Callable[[typing_extensions.CapsuleType], int]) → int`

See `SUNLinSolSetPreconditioner()`.

`sundials4py.core.SUNLinSolSetScalingVectors(S: sundials4py.core._generic_SUNLinearSolver, s1: sundials4py.core._generic_N_Vector, s2: sundials4py.core._generic_N_Vector) → int`



See *SUNLinSolSetScalingVectors()*.

`sundials4py.core.SUNLinSolSetZeroGuess(S: sundials4py.core._generic_SUNLinearSolver, onoff: int) → int`

See *SUNLinSolSetZeroGuess()*.

`sundials4py.core.SUNLinSolSetup(S: sundials4py.core._generic_SUNLinearSolver, A: sundials4py.core._generic_SUNMatrix | None = None) → int`

See *SUNLinSolSetup()*.

`sundials4py.core.SUNLinSolSolve(S: sundials4py.core._generic_SUNLinearSolver, A: sundials4py.core._generic_SUNMatrix | None, x: sundials4py.core._generic_N_Vector, b: sundials4py.core._generic_N_Vector, tol: float) → int`

See *SUNLinSolSolve()*.

`sundials4py.core.SUNLinSol_Band(y: sundials4py.core._generic_N_Vector, A: sundials4py.core._generic_SUNMatrix, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNLinearSolver`

`nb::keep_alive<0, 3>()`

See *SUNLinSol\_Band()*.

`sundials4py.core.SUNLinSol_Dense(y: sundials4py.core._generic_N_Vector, A: sundials4py.core._generic_SUNMatrix, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNLinearSolver`

`nb::keep_alive<0, 3>()`

See *SUNLinSol\_Dense()*.

`sundials4py.core.SUNLinSol_PCG(y: sundials4py.core._generic_N_Vector, pretype: int, maxl: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNLinearSolver`

`nb::keep_alive<0, 4>()`

See *SUNLinSol\_PCG()*.

`sundials4py.core.SUNLinSol_PCGSetMaxl(S: sundials4py.core._generic_SUNLinearSolver, maxl: int) → int`

See *SUNLinSol\_PCGSetMaxl()*.

`sundials4py.core.SUNLinSol_PCGSetPrecType(S: sundials4py.core._generic_SUNLinearSolver, pretype: int) → int`

See *SUNLinSol\_PCGSetPrecType()*.

`sundials4py.core.SUNLinSol_SPBCGS(y: sundials4py.core._generic_N_Vector, pretype: int, maxl: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNLinearSolver`

`nb::keep_alive<0, 4>()`

See *SUNLinSol\_SPBCGS()*.

`sundials4py.core.SUNLinSol_SPBCGSSetMaxl(S: sundials4py.core._generic_SUNLinearSolver, maxl: int) → int`

See *SUNLinSol\_SPBCGSSetMaxl()*.

`sundials4py.core.SUNLinSol_SPBCGSSetPrecType(S: sundials4py.core._generic_SUNLinearSolver, pretype: int) → int`

See `SUNLinSol_SPBCGSSetPrecType()`.

`sundials4py.core.SUNLinSol_SPFGMR(y: sundials4py.core._generic_N_Vector, pretype: int, maxl: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNLinearSolver`

`nb::keep_alive<0, 4>()`

See `SUNLinSol_SPFGMR()`.

`sundials4py.core.SUNLinSol_SPFGMRSetGSType(S: sundials4py.core._generic_SUNLinearSolver, gstype: int) → int`

See `SUNLinSol_SPFGMRSetGSType()`.

`sundials4py.core.SUNLinSol_SPFGMRSetMaxRestarts(S: sundials4py.core._generic_SUNLinearSolver, maxrs: int) → int`

See `SUNLinSol_SPFGMRSetMaxRestarts()`.

`sundials4py.core.SUNLinSol_SPFGMRSetPrecType(S: sundials4py.core._generic_SUNLinearSolver, pretype: int) → int`

See `SUNLinSol_SPFGMRSetPrecType()`.

`sundials4py.core.SUNLinSol_SPGMR(y: sundials4py.core._generic_N_Vector, pretype: int, maxl: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNLinearSolver`

`nb::keep_alive<0, 4>()`

See `SUNLinSol_SPGMR()`.

`sundials4py.core.SUNLinSol_SPGMRSetGSType(S: sundials4py.core._generic_SUNLinearSolver, gstype: int) → int`

See `SUNLinSol_SPGMRSetGSType()`.

`sundials4py.core.SUNLinSol_SPGMRSetMaxRestarts(S: sundials4py.core._generic_SUNLinearSolver, maxrs: int) → int`

See `SUNLinSol_SPGMRSetMaxRestarts()`.

`sundials4py.core.SUNLinSol_SPGMRSetPrecType(S: sundials4py.core._generic_SUNLinearSolver, pretype: int) → int`

See `SUNLinSol_SPGMRSetPrecType()`.

`sundials4py.core.SUNLinSol_SPTFQMR(y: sundials4py.core._generic_N_Vector, pretype: int, maxl: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNLinearSolver`

`nb::keep_alive<0, 4>()`

See `SUNLinSol_SPTFQMR()`.

`sundials4py.core.SUNLinSol_SPTFQMRSetMaxl(S: sundials4py.core._generic_SUNLinearSolver, maxl: int) → int`

See `SUNLinSol_SPTFQMRSetMaxl()`.

`sundials4py.core.SUNLinSol_SPTFQMRSetPrecType(S: sundials4py.core._generic_SUNLinearSolver, pretype: int) → int`

See `SUNLinSol_SPTFQMRSetPrecType()`.



`sundials4py.core.SUNLogErrHandlerFn`(*line: int, func: str, file: str, msg: str, err\_code: int, err\_user\_data: typing\_extensions.CapsuleType, suctx: sundials4py.core.SUNContext\_*) → None

See `SUNLogErrHandlerFn()`.

`sundials4py.core.SUNLogger_Create`(*comm: int, output\_rank: int*) → tuple[int, *sundials4py.core.SUNLogger\_*]

See `SUNLogger_Create()`.

`sundials4py.core.SUNLogger_CreateFromEnv`(*comm: int*) → tuple[int, *sundials4py.core.SUNLogger\_*]

See `SUNLogger_CreateFromEnv()`.

`sundials4py.core.SUNLogger_Flush`(*logger: sundials4py.core.SUNLogger\_, lvl: sundials4py.core.SUNLogLevel*) → int

See `SUNLogger_Flush()`.

`sundials4py.core.SUNLogger_GetOutputRank`(*logger: sundials4py.core.SUNLogger\_*) → tuple[int, int]

See `SUNLogger_GetOutputRank()`.

`sundials4py.core.SUNLogger_QueueMsg`(*logger: sundials4py.core.SUNLogger\_, lvl: sundials4py.core.SUNLogLevel, scope: str, label: str, msg\_txt: str*) → int

See `SUNLogger_QueueMsg()`.

`sundials4py.core.SUNLogger_SetDebugFilename`(*logger: sundials4py.core.SUNLogger\_, debug\_filename: str*) → int

See `SUNLogger_SetDebugFilename()`.

`sundials4py.core.SUNLogger_SetErrorFilename`(*logger: sundials4py.core.SUNLogger\_, error\_filename: str*) → int

See `SUNLogger_SetErrorFilename()`.

`sundials4py.core.SUNLogger_SetInfoFilename`(*logger: sundials4py.core.SUNLogger\_, info\_filename: str*) → int

See `SUNLogger_SetInfoFilename()`.

`sundials4py.core.SUNLogger_SetWarningFilename`(*logger: sundials4py.core.SUNLogger\_, warning\_filename: str*) → int

See `SUNLogger_SetWarningFilename()`.

`sundials4py.core.SUNMatClone`(*A: sundials4py.core.\_generic\_SUNMatrix*) → *sundials4py.core.\_generic\_SUNMatrix*

See `SUNMatClone()`.

`sundials4py.core.SUNMatCopy`(*A: sundials4py.core.\_generic\_SUNMatrix, B: sundials4py.core.\_generic\_SUNMatrix*) → int

See `SUNMatCopy()`.

`sundials4py.core.SUNMatGetID`(*A: sundials4py.core.\_generic\_SUNMatrix*) → *sundials4py.core.SUNMatrix\_ID*

See `SUNMatGetID()`.

`sundials4py.core.SUNMatHermitianTransposeVec`(*A: sundials4py.core.\_generic\_SUNMatrix, x: sundials4py.core.\_generic\_N\_Vector, y: sundials4py.core.\_generic\_N\_Vector*) → int

See `SUNMatHermitianTransposeVec()`.

`sundials4py.core.SUNMatMatvec`(*A*: `sundials4py.core._generic_SUNMatrix`, *x*:  
`sundials4py.core._generic_N_Vector`, *y*:  
`sundials4py.core._generic_N_Vector`) → `int`

See `SUNMatMatvec()`.

`sundials4py.core.SUNMatMatvecSetup`(*A*: `sundials4py.core._generic_SUNMatrix`) → `int`

See `SUNMatMatvecSetup()`.

`sundials4py.core.SUNMatScaleAdd`(*c*: `float`, *A*: `sundials4py.core._generic_SUNMatrix`, *B*:  
`sundials4py.core._generic_SUNMatrix`) → `int`

See `SUNMatScaleAdd()`.

`sundials4py.core.SUNMatScaleAddI`(*c*: `float`, *A*: `sundials4py.core._generic_SUNMatrix`) → `int`

See `SUNMatScaleAddI()`.

`sundials4py.core.SUNMatZero`(*A*: `sundials4py.core._generic_SUNMatrix`) → `int`

See `SUNMatZero()`.

`sundials4py.core.SUNMemoryHelper_Clone`(*param\_0*: `sundials4py.core.SUNMemoryHelper_`) →  
`sundials4py.core.SUNMemoryHelper_`

See `SUNMemoryHelper_Clone()`.

`sundials4py.core.SUNMemoryHelper_ImplementsRequiredOps`(*param\_0*:  
`sundials4py.core.SUNMemoryHelper_`) →  
`int`

See `SUNMemoryHelper_ImplementsRequiredOps()`.

`sundials4py.core.SUNMemoryHelper_SetDefaultQueue`(*param\_0*: `sundials4py.core.SUNMemoryHelper_`,  
*queue*: `typing_extensions.CapsuleType`) → `int`

See `SUNMemoryHelper_SetDefaultQueue()`.

`sundials4py.core.SUNMemoryHelper_Sys`(*sunctx*: `sundials4py.core.SUNContext_`) →  
`sundials4py.core.SUNMemoryHelper_`

`nb::keep_alive<0, 1>()`

See `SUNMemoryHelper_Sys()`.

`sundials4py.core.SUNModifiedGS`(*v\_1d*: `collections.abc.Sequence[sundials4py.core._generic_N_Vector]`,  
*h\_2d*: `numpy.ndarray[dtype=float64, shape=(*, order='C'), k: int, p: int]`  
→ `tuple[int, float]`

See `SUNModifiedGS()`.

`sundials4py.core.SUNNonlinSolGetCurIter`(*NLS*: `sundials4py.core._generic_SUNNonlinearSolver`) →  
`tuple[int, int]`

See `SUNNonlinSolGetCurIter()`.

`sundials4py.core.SUNNonlinSolGetNumConvFails`(*NLS*: `sundials4py.core._generic_SUNNonlinearSolver`)  
→ `tuple[int, int]`

See `SUNNonlinSolGetNumConvFails()`.

`sundials4py.core.SUNNonlinSolGetNumIters`(*NLS*: `sundials4py.core._generic_SUNNonlinearSolver`) →  
`tuple[int, int]`

See `SUNNonlinSolGetNumIters()`.

`sundials4py.core.SUNNonlinSolGetType`(*NLS*: `sundials4py.core._generic_SUNNonlinearSolver`) →  
`sundials4py.core.SUNNonlinearSolver_Type`

See `SUNNonlinSolGetType()`.

`sundials4py.core.SUNNonlinSolInitialize(NLS: sundials4py.core._generic_SUNNonlinearSolver) → int`  
 See `SUNNonlinSolInitialize()`.

`sundials4py.core.SUNNonlinSolSetConvTestFn(NLS: sundials4py.core._generic_SUNNonlinearSolver, CTestFn: collections.abc.Callable[[sundials4py.core._generic_SUNNonlinearSolver, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, float, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`  
 See `SUNNonlinSolSetConvTestFn()`.

`sundials4py.core.SUNNonlinSolSetDamping_FixedPoint(NLS: sundials4py.core._generic_SUNNonlinearSolver, beta: float) → int`  
 See `SUNNonlinSolSetDamping_FixedPoint()`.

`sundials4py.core.SUNNonlinSolSetLSetupFn(NLS: sundials4py.core._generic_SUNNonlinearSolver, SetupFn: collections.abc.Callable[[int, typing_extensions.CapsuleType], tuple[int, int]] | None) → int`  
 See `SUNNonlinSolSetLSetupFn()`.

`sundials4py.core.SUNNonlinSolSetLSolveFn(NLS: sundials4py.core._generic_SUNNonlinearSolver, SolveFn: collections.abc.Callable[[sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`  
 See `SUNNonlinSolSetLSolveFn()`.

`sundials4py.core.SUNNonlinSolSetMaxIters(NLS: sundials4py.core._generic_SUNNonlinearSolver, maxiters: int) → int`  
 See `SUNNonlinSolSetMaxIters()`.

`sundials4py.core.SUNNonlinSolSetOptions(self: sundials4py.core._generic_SUNNonlinearSolver, id: str, file_name: str, argc: int, args: collections.abc.Sequence[str]) → int`  
 See `SUNNonlinSolSetOptions()`.

`sundials4py.core.SUNNonlinSolSetSysFn(NLS: sundials4py.core._generic_SUNNonlinearSolver, SysFn: collections.abc.Callable[[sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`  
 See `SUNNonlinSolSetSysFn()`.

`sundials4py.core.SUNNonlinSolSetup(NLS: sundials4py.core._generic_SUNNonlinearSolver, y: sundials4py.core._generic_N_Vector) → int`  
 See `SUNNonlinSolSetup()`.

`sundials4py.core.SUNNonlinSolSolve(NLS: sundials4py.core._generic_SUNNonlinearSolver, y0: sundials4py.core._generic_N_Vector, y: sundials4py.core._generic_N_Vector, w: sundials4py.core._generic_N_Vector, tol: float, callLSetup: int) → int`  
 See `SUNNonlinSolSolve()`.

`sundials4py.core.SUNNonlinSol_FixedPoint(y: sundials4py.core._generic_N_Vector, m: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNNonlinearSolver`

`nb::keep_alive<0, 3>()`

See `SUNNonlinSol_FixedPoint()`.

`sundials4py.core.SUNNonlinSol_FixedPointSens(count: int, y: sundials4py.core._generic_N_Vector, m: int, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNNonlinearSolver`

`nb::keep_alive<0, 4>()`

See `SUNNonlinSol_FixedPointSens()`.

`sundials4py.core.SUNNonlinSol_Newton(y: sundials4py.core._generic_N_Vector, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNNonlinearSolver`

`nb::keep_alive<0, 2>()`

See `SUNNonlinSol_Newton()`.

`sundials4py.core.SUNNonlinSol_NewtonSens(count: int, y: sundials4py.core._generic_N_Vector, sunctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNNonlinearSolver`

`nb::keep_alive<0, 3>()`

See `SUNNonlinSol_NewtonSens()`.

`sundials4py.core.SUNProfiler_Begin(p: sundials4py.core.SUNProfiler_, name: str) → int`  
See `SUNProfiler_Begin()`.

`sundials4py.core.SUNProfiler_Create(comm: int, title: str) → tuple[int, sundials4py.core.SUNProfiler_]`  
See `SUNProfiler_Create()`.

`sundials4py.core.SUNProfiler_End(p: sundials4py.core.SUNProfiler_, name: str) → int`  
See `SUNProfiler_End()`.

`sundials4py.core.SUNProfiler_GetElapsedTime(p: sundials4py.core.SUNProfiler_, name: str) → tuple[int, float]`  
See `SUNProfiler_GetElapsedTime()`.

`sundials4py.core.SUNProfiler_GetTimerResolution(p: sundials4py.core.SUNProfiler_) → tuple[int, float]`  
See `SUNProfiler_GetTimerResolution()`.

`sundials4py.core.SUNProfiler_Print(p: sundials4py.core.SUNProfiler_, fp: sundials4py.core.FILE) → int`  
See `SUNProfiler_Print()`.

`sundials4py.core.SUNProfiler_Reset(p: sundials4py.core.SUNProfiler_) → int`  
See `SUNProfiler_Reset()`.

`sundials4py.core.SUNQRAdd_CGS2(Q_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], R_1d: numpy.ndarray[dtype=float64, shape=(*,), order='C'], df: sundials4py.core._generic_N_Vector, m: int, mMax: int, QRdata: typing_extensions.CapsuleType) → int`

See `SUNQRAdd_CGS2()`.

`sundials4py.core.SUNQRAdd_DCGS2(Q_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], R_1d: numpy.ndarray[dtype=float64, shape=(*,), order='C'], df: sundials4py.core._generic_N_Vector, m: int, mMax: int, QRdata: typing_extensions.CapsuleType) → int`

See `SUNQRAdd_DCGS2()`.

`sundials4py.core.SUNQRAdd_DCGS2_SB(Q_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], R_1d: numpy.ndarray[dtype=float64, shape=(*), order='C'], df: sundials4py.core._generic_N_Vector, m: int, mMax: int, QRdata: typing_extensions.CapsuleType) → int`

See `SUNQRAdd_DCGS2_SB()`.

`sundials4py.core.SUNQRAdd_ICWY(Q_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], R_1d: numpy.ndarray[dtype=float64, shape=(*), order='C'], df: sundials4py.core._generic_N_Vector, m: int, mMax: int, QRdata: typing_extensions.CapsuleType) → int`

See `SUNQRAdd_ICWY()`.

`sundials4py.core.SUNQRAdd_ICWY_SB(Q_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], R_1d: numpy.ndarray[dtype=float64, shape=(*), order='C'], df: sundials4py.core._generic_N_Vector, m: int, mMax: int, QRdata: typing_extensions.CapsuleType) → int`

See `SUNQRAdd_ICWY_SB()`.

`sundials4py.core.SUNQRAdd_MGS(Q_1d: collections.abc.Sequence[sundials4py.core._generic_N_Vector], R_1d: numpy.ndarray[dtype=float64, shape=(*), order='C'], df: sundials4py.core._generic_N_Vector, m: int, mMax: int, QRdata: typing_extensions.CapsuleType) → int`

See `SUNQRAdd_MGS()`.

`sundials4py.core.SUNQRfact(n: int, h_2d: numpy.ndarray[dtype=float64, shape=(*), order='C'], q_1d: numpy.ndarray[dtype=float64, shape=(*), order='C'], job: int) → int`

See `SUNQRfact()`.

`sundials4py.core.SUNQRsol(n: int, h_2d: numpy.ndarray[dtype=float64, shape=(*), order='C'], q_1d: numpy.ndarray[dtype=float64, shape=(*), order='C'], b_1d: numpy.ndarray[dtype=float64, shape=(*), order='C']) → int`

See `SUNQRsol()`.

`sundials4py.core.SUNSparseFromBandMatrix(A: sundials4py.core._generic_SUNMatrix, droptol: float, sparsetype: int) → sundials4py.core._generic_SUNMatrix`

See `SUNSparseFromBandMatrix()`.

`sundials4py.core.SUNSparseFromDenseMatrix(A: sundials4py.core._generic_SUNMatrix, droptol: float, sparsetype: int) → sundials4py.core._generic_SUNMatrix`

See `SUNSparseFromDenseMatrix()`.

`sundials4py.core.SUNSparseMatrix(M: int, N: int, NNZ: int, sparsetype: int, suncctx: sundials4py.core.SUNContext_) → sundials4py.core._generic_SUNMatrix`

`nb::keep_alive<0, 5>()`

See `SUNSparseMatrix()`.

`sundials4py.core.SUNSparseMatrix_Columns(A: sundials4py.core._generic_SUNMatrix) → int`

See `SUNSparseMatrix_Columns()`.

`sundials4py.core.SUNSparseMatrix_Data(A: sundials4py.core._generic_SUNMatrix) → numpy.ndarray[dtype=float64, shape=(*), order='C']`

See `SUNSparseMatrix_Data()`.

`sundials4py.core.SUNSparseMatrix_IndexPointers`(A: `sundials4py.core._generic_SUNMatrix`) →  
numpy.ndarray[`dtype=int64, shape=(*)`, `order='C'`]

See `SUNSparseMatrix_IndexPointers()`.

`sundials4py.core.SUNSparseMatrix_IndexValues`(A: `sundials4py.core._generic_SUNMatrix`) →  
numpy.ndarray[`dtype=int64, shape=(*)`, `order='C'`]

See `SUNSparseMatrix_IndexValues()`.

`sundials4py.core.SUNSparseMatrix_NNZ`(A: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNSparseMatrix_NNZ()`.

`sundials4py.core.SUNSparseMatrix_NP`(A: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNSparseMatrix_NP()`.

`sundials4py.core.SUNSparseMatrix_Print`(A: `sundials4py.core._generic_SUNMatrix`, *outfile*:  
`sundials4py.core.FILE`) → None

See `SUNSparseMatrix_Print()`.

`sundials4py.core.SUNSparseMatrix_Realloc`(A: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNSparseMatrix_Realloc()`.

`sundials4py.core.SUNSparseMatrix_Reallocate`(A: `sundials4py.core._generic_SUNMatrix`, *NNZ*: int) → int

See `SUNSparseMatrix_Reallocate()`.

`sundials4py.core.SUNSparseMatrix_Rows`(A: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNSparseMatrix_Rows()`.

`sundials4py.core.SUNSparseMatrix_SparseType`(A: `sundials4py.core._generic_SUNMatrix`) → int

See `SUNSparseMatrix_SparseType()`.

`sundials4py.core.SUNSparseMatrix_ToCSC`(A: `sundials4py.core._generic_SUNMatrix`) → tuple[int,  
`sundials4py.core._generic_SUNMatrix`]

See `SUNSparseMatrix_ToCSC()`.

`sundials4py.core.SUNSparseMatrix_ToCSR`(A: `sundials4py.core._generic_SUNMatrix`) → tuple[int,  
`sundials4py.core._generic_SUNMatrix`]

See `SUNSparseMatrix_ToCSR()`.

`sundials4py.core.SUNStepper_Create`(*sunctx*: `sundials4py.core.SUNContext_`) → tuple[int,  
`sundials4py.core.SUNStepper_`]

`nb::call_policy<sundials4py::returns_references_to<1, 1>>()`

See `SUNStepper_Create()`.

`sundials4py.core.SUNStepper_Evolve`(*stepper*: `sundials4py.core.SUNStepper_`, *tout*: float, *vret*:  
`sundials4py.core._generic_N_Vector`) → tuple[int, float]

See `SUNStepper_Evolve()`.

`sundials4py.core.SUNStepper_FullRhs`(*stepper*: `sundials4py.core.SUNStepper_`, *t*: float, *v*:  
`sundials4py.core._generic_N_Vector`, *f*:  
`sundials4py.core._generic_N_Vector`, *mode*:  
`sundials4py.core.SUNFullRhsMode`) → int

See `SUNStepper_FullRhs()`.

`sundials4py.core.SUNStepper_GetLastFlag`(*stepper*: `sundials4py.core.SUNStepper_`) → tuple[int, int]

See `SUNStepper_GetLastFlag()`.



`sundials4py.core.SUNStepper_GetNumSteps`(*stepper*: `sundials4py.core.SUNStepper_`) → `tuple[int, int]`

See [`SUNStepper\_GetNumSteps\(\)`](#).

`sundials4py.core.SUNStepper_OneStep`(*stepper*: `sundials4py.core.SUNStepper_`, *tout*: `float`, *vret*: `sundials4py.core._generic_N_Vector`) → `tuple[int, float]`

See [`SUNStepper\_OneStep\(\)`](#).

`sundials4py.core.SUNStepper_ReInit`(*stepper*: `sundials4py.core.SUNStepper_`, *t0*: `float`, *v0*: `sundials4py.core._generic_N_Vector`) → `int`

See [`SUNStepper\_ReInit\(\)`](#).

`sundials4py.core.SUNStepper_Reset`(*stepper*: `sundials4py.core.SUNStepper_`, *tR*: `float`, *vR*: `sundials4py.core._generic_N_Vector`) → `int`

See [`SUNStepper\_Reset\(\)`](#).

`sundials4py.core.SUNStepper_ResetCheckpointIndex`(*stepper*: `sundials4py.core.SUNStepper_`, *ckptIdxR*: `int`) → `int`

See [`SUNStepper\_ResetCheckpointIndex\(\)`](#).

`sundials4py.core.SUNStepper_SetEvolveFn`(*stepper*: `sundials4py.core.SUNStepper_`, *fn*: `collections.abc.Callable[[sundials4py.core.SUNStepper_, float, sundials4py.core._generic_N_Vector, float], int] | None`) → `int`

See [`SUNStepper\_SetEvolveFn\(\)`](#).

`sundials4py.core.SUNStepper_SetForcing`(*stepper*: `sundials4py.core.SUNStepper_`, *tshift*: `float`, *tscale*: `float`, *forcing\_id*: `collections.abc.Sequence[sundials4py.core._generic_N_Vector]`, *nforcing*: `int`) → `int`

See [`SUNStepper\_SetForcing\(\)`](#).

`sundials4py.core.SUNStepper_SetForcingFn`(*stepper*: `sundials4py.core.SUNStepper_`, *fn*: `collections.abc.Callable[[sundials4py.core.SUNStepper_, float, float, collections.abc.Sequence[sundials4py.core._generic_N_Vector], int], int] | None`) → `int`

See [`SUNStepper\_SetForcingFn\(\)`](#).

`sundials4py.core.SUNStepper_SetFullRhsFn`(*stepper*: `sundials4py.core.SUNStepper_`, *fn*: `collections.abc.Callable[[sundials4py.core.SUNStepper_, float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core.SUNFullRhsMode], int] | None`) → `int`

See [`SUNStepper\_SetFullRhsFn\(\)`](#).

`sundials4py.core.SUNStepper_SetGetNumStepsFn`(*stepper*: `sundials4py.core.SUNStepper_`, *fn*: `collections.abc.Callable[[sundials4py.core.SUNStepper_, tuple[int, int]] | None`) → `int`

See [`SUNStepper\_SetGetNumStepsFn\(\)`](#).

`sundials4py.core.SUNStepper_SetLastFlag`(*stepper*: `sundials4py.core.SUNStepper_`, *last\_flag*: `int`) → `int`

See [`SUNStepper\_SetLastFlag\(\)`](#).

`sundials4py.core.SUNStepper_SetOneStepFn`(*stepper*: `sundials4py.core.SUNStepper_`, *fn*: `collections.abc.Callable[[sundials4py.core.SUNStepper_, float, sundials4py.core._generic_N_Vector, float], int] | None`) → `int`

See `SUNStepper_SetOneStepFn()`.

```
sundials4py.core.SUNStepper_SetReInitFn(stepper: sundials4py.core.SUNStepper_, fn:  
collections.abc.Callable[[sundials4py.core.SUNStepper_, float,  
sundials4py.core._generic_N_Vector], int] | None) → int
```

See `SUNStepper_SetReInitFn()`.

```
sundials4py.core.SUNStepper_SetResetCheckpointIndexFn(stepper: sundials4py.core.SUNStepper_, fn:  
collections.abc.Callable[[sundials4py.core.SUNStepper_,  
int], int] | None) →  
int
```

See `SUNStepper_SetResetCheckpointIndexFn()`.

```
sundials4py.core.SUNStepper_SetResetFn(stepper: sundials4py.core.SUNStepper_, fn:  
collections.abc.Callable[[sundials4py.core.SUNStepper_, float,  
sundials4py.core._generic_N_Vector], int] | None) → int
```

See `SUNStepper_SetResetFn()`.

```
sundials4py.core.SUNStepper_SetStepDirection(stepper: sundials4py.core.SUNStepper_, stepdir: float)  
→ int
```

See `SUNStepper_SetStepDirection()`.

```
sundials4py.core.SUNStepper_SetStepDirectionFn(stepper: sundials4py.core.SUNStepper_, fn: collec-  
tions.abc.Callable[[sundials4py.core.SUNStepper_,  
float], int] | None) → int
```

See `SUNStepper_SetStepDirectionFn()`.

```
sundials4py.core.SUNStepper_SetStopTime(stepper: sundials4py.core.SUNStepper_, tstop: float) → int
```

See `SUNStepper_SetStopTime()`.

```
sundials4py.core.SUNStepper_SetStopTimeFn(stepper: sundials4py.core.SUNStepper_, fn:  
collections.abc.Callable[[sundials4py.core.SUNStepper_,  
float], int] | None) → int
```

See `SUNStepper_SetStopTimeFn()`.

## 21.3 arkode Submodule

### 21.3.1 Classes

A submodule of ‘sundials4py’

```
class sundials4py.arkode.ARKAccumError(*values)
```

```
    ARK_ACCUMERROR_AVG = 3
```

```
    ARK_ACCUMERROR_MAX = 1
```

```
    ARK_ACCUMERROR_NONE = 0
```

```
    ARK_ACCUMERROR_SUM = 2
```

```
class sundials4py.arkode.ARKODE_DIRKTableID(*values)
```



```

ARKODE_ARK2_DIRK_3_1_2 = 123
ARKODE_ARK324L2SA_DIRK_4_2_3 = 104
ARKODE_ARK436L2SA_DIRK_6_3_4 = 109
ARKODE_ARK437L2SA_DIRK_7_3_4 = 112
ARKODE_ARK548L2SA_DIRK_8_4_5 = 111
ARKODE_ARK548L2SAb_DIRK_8_4_5 = 113
ARKODE_BACKWARD_EULER_1_1 = 124
ARKODE_BILLINGTON_3_3_2 = 101
ARKODE_CASH_5_2_4 = 105
ARKODE_CASH_5_3_4 = 106
ARKODE_DIRK_NONE = -1
ARKODE_ESDIRK324L2SA_4_2_3 = 114
ARKODE_ESDIRK325L2SA_5_2_3 = 115
ARKODE_ESDIRK32I5L2SA_5_2_3 = 116
ARKODE_ESDIRK436L2SA_6_3_4 = 117
ARKODE_ESDIRK437L2SA_7_3_4 = 120
ARKODE_ESDIRK43I6L2SA_6_3_4 = 118
ARKODE_ESDIRK547L2SA2_7_4_5 = 122
ARKODE_ESDIRK547L2SA_7_4_5 = 121
ARKODE_IMPLICIT_MIDPOINT_1_2 = 125
ARKODE_IMPLICIT_TRAPEZOIDAL_2_2 = 126
ARKODE_KVAERNO_4_2_3 = 103
ARKODE_KVAERNO_5_3_4 = 108
ARKODE_KVAERNO_7_4_5 = 110
ARKODE_MAX_DIRK_NUM = 126
ARKODE_MIN_DIRK_NUM = 100
ARKODE_QESDIRK436L2SA_6_3_4 = 119
ARKODE_SDIRK_2_1_2 = 100
ARKODE_SDIRK_5_3_4 = 107
ARKODE_TRBDF2_3_3_2 = 102

```

```
class sundials4py.arkode.ARKODE_ERKTableID(*values)
```

```
ARKODE_ARK2_ERK_3_1_2 = 15
ARKODE_ARK324L2SA_ERK_4_2_3 = 2
ARKODE_ARK436L2SA_ERK_6_3_4 = 4
ARKODE_ARK437L2SA_ERK_7_3_4 = 13
ARKODE_ARK548L2SA_ERK_8_4_5 = 9
ARKODE_ARK548L2SAb_ERK_8_4_5 = 14
ARKODE_BOGACKI_SHAMPINE_4_2_3 = 1
ARKODE_CASH_KARP_6_4_5 = 6
ARKODE_DORMAND_PRINCE_7_4_5 = 8
ARKODE_ERK_NONE = -1
ARKODE_EXPLICIT_MIDPOINT_EULER_2_1_2 = 24
ARKODE_FEHLBERG_13_7_8 = 11
ARKODE_FEHLBERG_6_4_5 = 7
ARKODE_FORWARD_EULER_1_1 = 22
ARKODE_HEUN_EULER_2_1_2 = 0
ARKODE_KNOTH_WOLKE_3_3 = 12
ARKODE_MAX_ERK_NUM = 26
ARKODE_MIN_ERK_NUM = 0
ARKODE_RALSTON_3_1_2 = 25
ARKODE_RALSTON_EULER_2_1_2 = 23
ARKODE_SAYFY_ABURUB_6_3_4 = 5
ARKODE_SHU_OSHER_3_2_3 = 17
ARKODE_SOFRONIOU_SPALETTA_5_3_4 = 16
ARKODE_TSITOURAS_7_4_5 = 26
ARKODE_VERNER_10_6_7 = 19
ARKODE_VERNER_13_7_8 = 20
ARKODE_VERNER_16_8_9 = 21
ARKODE_VERNER_8_5_6 = 10
ARKODE_VERNER_9_5_6 = 18
ARKODE_ZONNEVELD_5_3_4 = 3
```

```
class sundials4py.arkode.ARKODE_LSRKMethodType(*values)
```

```
ARKODE_LSRK_RKC_2 = 0
```

```
ARKODE_LSRK_RKL_2 = 1
```

```
ARKODE_LSRK_SSP_10_4 = 4
```

```
ARKODE_LSRK_SSP_S_2 = 2
```

```
ARKODE_LSRK_SSP_S_3 = 3
```

```
class sundials4py.arkode.ARKODE_MRITableID(*values)
```

```
    MRI coupling table IDs
```

```
ARKODE_IMEX_MRI_GARK3a = 206
```

```
ARKODE_IMEX_MRI_GARK3b = 207
```

```
ARKODE_IMEX_MRI_GARK4 = 208
```

```
ARKODE_IMEX_MRI_GARK_EULER = 216
```

```
ARKODE_IMEX_MRI_GARK_MIDPOINT = 218
```

```
ARKODE_IMEX_MRI_GARK_TRAPEZOIDAL = 217
```

```
ARKODE_IMEX_MRI_SR21 = 223
```

```
ARKODE_IMEX_MRI_SR32 = 224
```

```
ARKODE_IMEX_MRI_SR43 = 225
```

```
ARKODE_MAX_MRI_NUM = 225
```

```
ARKODE_MERK21 = 219
```

```
ARKODE_MERK32 = 220
```

```
ARKODE_MERK43 = 221
```

```
ARKODE_MERK54 = 222
```

```
ARKODE_MIN_MRI_NUM = 200
```

```
ARKODE_MIS_KW3 = 200
```

```
ARKODE_MRI_GARK_BACKWARD_EULER = 214
```

```
ARKODE_MRI_GARK_ERK22a = 211
```

```
ARKODE_MRI_GARK_ERK22b = 212
```

```
ARKODE_MRI_GARK_ERK33a = 201
```

```
ARKODE_MRI_GARK_ERK45a = 202
```

```
ARKODE_MRI_GARK_ESDIRK34a = 204
```

```
ARKODE_MRI_GARK_ESDIRK46a = 205
```

```
ARKODE_MRI_GARK_FORWARD_EULER = 209
```

```
ARKODE_MRI_GARK_IMPLICIT_MIDPOINT = 215
```

```
ARKODE_MRI_GARK_IRK21a = 203
```

```
ARKODE_MRI_GARK_RALSTON2 = 210
```

```
ARKODE_MRI_GARK_RALSTON3 = 213
```

```
ARKODE_MRI_NONE = -1
```

```
class sundials4py.arkode.ARKODE_SPRKMethodID(*values)
```

```
    ARKODE_MAX_SPRK_NUM = 11
```

```
    ARKODE_MIN_SPRK_NUM = 0
```

```
    ARKODE_SPRK_CANDY_ROZMUS_4_4 = 6
```

```
    ARKODE_SPRK_EULER_1_1 = 0
```

```
    ARKODE_SPRK_LEAPFROG_2_2 = 1
```

```
    ARKODE_SPRK_MCLACHLAN_2_2 = 4
```

```
    ARKODE_SPRK_MCLACHLAN_3_3 = 5
```

```
    ARKODE_SPRK_MCLACHLAN_4_4 = 7
```

```
    ARKODE_SPRK_MCLACHLAN_5_6 = 8
```

```
    ARKODE_SPRK_NONE = -1
```

```
    ARKODE_SPRK_PSEUDO_LEAPFROG_2_2 = 2
```

```
    ARKODE_SPRK_RUTH_3_3 = 3
```

```
    ARKODE_SPRK_SOFRONIOU_10_36 = 11
```

```
    ARKODE_SPRK_SUZUKI_UMENO_8_16 = 10
```

```
    ARKODE_SPRK_YOSHIDA_6_8 = 9
```

```
class sundials4py.arkode.ARKODE_SplittingCoefficientsID(*values)
```

```
    ARKODE_MAX_SPLITTING_NUM = 6
```

```
    ARKODE_MIN_SPLITTING_NUM = 0
```

```
    ARKODE_SPLITTING_BEST_2_2_2 = 2
```

```
    ARKODE_SPLITTING_LIE_TROTTER_1_1_2 = 0
```

```
    ARKODE_SPLITTING_NONE = -1
```

```
    ARKODE_SPLITTING_RUTH_3_3_2 = 4
```

```
    ARKODE_SPLITTING_STRANG_2_2_2 = 1
```

```
    ARKODE_SPLITTING_SUZUKI_3_3_2 = 3
```

```
    ARKODE_SPLITTING_YOSHIDA_4_4_2 = 5
```

```

ARKODE_SPLITTING_YOSHIDA_8_6_2 = 6

class sundials4py.arkode.ARKRelaxSolver(*values)
    ARK_RELAX_BRENT = 0
    ARK_RELAX_NEWTON = 1

class sundials4py.arkode.ARKodeButcherTableMem(*args, **kwargs)

class sundials4py.arkode.ARKodeSPRKTableMem(*args, **kwargs)

class sundials4py.arkode.ARKodeView
    get

class sundials4py.arkode.MRISTEP_METHOD_TYPE(*values)
    MRISTEP_EXPLICIT = 0
    MRISTEP_IMEX = 2
    MRISTEP_IMPLICIT = 1
    MRISTEP_MERK = 3
    MRISTEP_SR = 4

class sundials4py.arkode.MRIStepCouplingMem(*args, **kwargs)

class sundials4py.arkode.SplittingStepCoefficientsMem(*args, **kwargs)

class sundials4py.arkode._MRIStepInnerStepper

```

### 21.3.2 Functions

`sundials4py.arkode.ARKStepCreate`(*fe*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*] | *None*, *fi*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*] | *None*, *t0*: *float*, *y0*: *sundials4py.core.\_generic\_N\_Vector*, *sunctx*: *sundials4py.core.SUNContext\_*) → *sundials4py.arkode.ARKodeView*

See [ARKStepCreate\(\)](#).

`sundials4py.arkode.ARKStepCreateAdjointStepper`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *adj\_fe*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*] | *None*, *adj\_fi*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*] | *None*, *tf*: *float*, *sf*: *sundials4py.core.\_generic\_N\_Vector*, *sunctx*: *sundials4py.core.SUNContext\_*) → *tuple*[*int*, *sundials4py.core.SUNAdjointStepper\_*]

See [ARKStepCreateAdjointStepper\(\)](#).

`sundials4py.arkode.ARKStepGetCurrentButcherTables(arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.arkode.ARKodeButcherTableMem, sundials4py.arkode.ARKodeButcherTableMem]`

Optional output functions

`nb::rv_policy::reference`

See [ARKStepGetCurrentButcherTables\(\)](#).

`sundials4py.arkode.ARKStepGetTimestepperStats(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int, int, int, int, int, int, int]`

See [ARKStepGetTimestepperStats\(\)](#).

`sundials4py.arkode.ARKStepSetExplicit(arkode_mem: typing_extensions.CapsuleType) → int`

See [ARKStepSetExplicit\(\)](#).

`sundials4py.arkode.ARKStepSetImEx(arkode_mem: typing_extensions.CapsuleType) → int`

See [ARKStepSetImEx\(\)](#).

`sundials4py.arkode.ARKStepSetImplicit(arkode_mem: typing_extensions.CapsuleType) → int`

See [ARKStepSetImplicit\(\)](#).

`sundials4py.arkode.ARKStepSetTableName(arkode_mem: typing_extensions.CapsuleType, itable: str, etable: str) → int`

See [ARKStepSetTableName\(\)](#).

`sundials4py.arkode.ARKStepSetTableNum(arkode_mem: typing_extensions.CapsuleType, itable: sundials4py.arkode.ARKODE_DIRKTableID, etable: sundials4py.arkode.ARKODE_ERKTableID) → int`

See [ARKStepSetTableNum\(\)](#).

`sundials4py.arkode.ARKStepSetTables(arkode_mem: typing_extensions.CapsuleType, q: int, p: int, Bi: sundials4py.arkode.ARKodeButcherTableMem, Be: sundials4py.arkode.ARKodeButcherTableMem) → int`

See [ARKStepSetTables\(\)](#).

`sundials4py.arkode.ARKodeButcherTable_CheckARKOrder(B1: sundials4py.arkode.ARKodeButcherTableMem, B2: sundials4py.arkode.ARKodeButcherTableMem, outfile: sundials4py.core.FILE) → tuple[int, int, int]`

See [ARKodeButcherTable\\_CheckARKOrder\(\)](#).

`sundials4py.arkode.ARKodeButcherTable_CheckOrder(B: sundials4py.arkode.ARKodeButcherTableMem, outfile: sundials4py.core.FILE) → tuple[int, int, int]`

See [ARKodeButcherTable\\_CheckOrder\(\)](#).

`sundials4py.arkode.ARKodeButcherTable_Copy(B: sundials4py.arkode.ARKodeButcherTableMem) → sundials4py.arkode.ARKodeButcherTableMem`

See [ARKodeButcherTable\\_Copy\(\)](#).

`sundials4py.arkode.ARKodeButcherTable_Create`(*s*: int, *q*: int, *p*: int, *c\_1d*: `numpy.ndarray`[dtype=float64, shape=(\*)], *order*='C', *A\_1d*: `numpy.ndarray`[dtype=float64, shape=(\*)], *order*='C', *b\_1d*: `numpy.ndarray`[dtype=float64, shape=(\*)], *order*='C', *d\_1d*: `numpy.ndarray`[dtype=float64, shape=(\*)], *order*='C']) → `sundials4py.arkode.ARKodeButcherTableMem`

See `ARKodeButcherTable_Create()`.

`sundials4py.arkode.ARKodeButcherTable_DIRKIDToName`(*imethod*: `sundials4py.arkode.ARKODE_DIRKTableID`) → str

See `ARKodeButcherTable_DIRKIDToName()`.

`sundials4py.arkode.ARKodeButcherTable_ERKIDToName`(*emethod*: `sundials4py.arkode.ARKODE_ERKTableID`) → str

See `ARKodeButcherTable_ERKIDToName()`.

`sundials4py.arkode.ARKodeButcherTable_IsStifflyAccurate`(*B*: `sundials4py.arkode.ARKodeButcherTableMem`) → int

See `ARKodeButcherTable_IsStifflyAccurate()`.

`sundials4py.arkode.ARKodeButcherTable_LoadDIRK`(*imethod*: `sundials4py.arkode.ARKODE_DIRKTableID`) → `sundials4py.arkode.ARKodeButcherTableMem`

Accessor routine to load built-in DIRK table

See `ARKodeButcherTable_LoadDIRK()`.

`sundials4py.arkode.ARKodeButcherTable_LoadDIRKByName`(*imethod*: str) → `sundials4py.arkode.ARKodeButcherTableMem`

Accessor routine to load built-in DIRK table

See `ARKodeButcherTable_LoadDIRKByName()`.

`sundials4py.arkode.ARKodeButcherTable_LoadERK`(*emethod*: `sundials4py.arkode.ARKODE_ERKTableID`) → `sundials4py.arkode.ARKodeButcherTableMem`

Accessor routine to load built-in ERK table

See `ARKodeButcherTable_LoadERK()`.

`sundials4py.arkode.ARKodeButcherTable_LoadERKByName`(*emethod*: str) → `sundials4py.arkode.ARKodeButcherTableMem`

See `ARKodeButcherTable_LoadERKByName()`.

`sundials4py.arkode.ARKodeButcherTable_Write`(*B*: `sundials4py.arkode.ARKodeButcherTableMem`, *outfile*: `sundials4py.core.FILE`) → None

See `ARKodeButcherTable_Write()`.

`sundials4py.arkode.ARKodeClearStopTime`(*arkode\_mem*: `typing_extensions.CapsuleType`) → int

See `ARKodeClearStopTime()`.

`sundials4py.arkode.ARKodeComputeState`(*arkode\_mem*: `typing_extensions.CapsuleType`, *zcor*: `sundials4py.core._generic_N_Vector`, *z*: `sundials4py.core._generic_N_Vector`) → int

Utility function to update/compute y based on zcor

See [ARKodeComputeState\(\)](#).

`sundials4py.arkode.ARKodeCreateMRISetInnerStepper(arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.arkode._MRISetInnerStepper]`

`sundials4py.arkode.ARKodeCreateMRISetInnerStepper(inner_arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.arkode._MRISetInnerStepper]`

Overloaded function.

1. `ARKodeCreateMRISetInnerStepper(arkode_mem: typing_extensions.CapsuleType) -> tuple[int, sundials4py.arkode._MRISetInnerStepper]`

Utility to wrap ARKODE as an MRISetInnerStepper

2. `ARKodeCreateMRISetInnerStepper(inner_arkode_mem: typing_extensions.CapsuleType) -> tuple[int, sundials4py.arkode._MRISetInnerStepper]`

See [ARKodeCreateMRISetInnerStepper\(\)](#).

`sundials4py.arkode.ARKodeCreateSUNStepper(arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.core.SUNStepper_]`

SUNStepper functions

See [ARKodeCreateSUNStepper\(\)](#).

`sundials4py.arkode.ARKodeEvolve(arkode_mem: typing_extensions.CapsuleType, tout: float, yout: sundials4py.core._generic_N_Vector, itask: int) → tuple[int, float]`

Integrate the ODE over an interval in t

See [ARKodeEvolve\(\)](#).

`sundials4py.arkode.ARKodeGetAccumulatedError(arkode_mem: typing_extensions.CapsuleType) → tuple[int, float]`

See [ARKodeGetAccumulatedError\(\)](#).

`sundials4py.arkode.ARKodeGetActualInitStep(arkode_mem: typing_extensions.CapsuleType) → tuple[int, float]`

See [ARKodeGetActualInitStep\(\)](#).

`sundials4py.arkode.ARKodeGetCurrentGamma(arkode_mem: typing_extensions.CapsuleType) → tuple[int, float]`

See [ARKodeGetCurrentGamma\(\)](#).

`sundials4py.arkode.ARKodeGetCurrentMassMatrix(arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.core._generic_SUNMatrix]`

Optional output functions (non-identity mass matrices)

`nb::rv_policy::reference`

See [ARKodeGetCurrentMassMatrix\(\)](#).

`sundials4py.arkode.ARKodeGetCurrentState(arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.core._generic_N_Vector]`

`nb::rv_policy::reference`

See [ARKodeGetCurrentState\(\)](#).



`sundials4py.arkode.ARKodeGetCurrentStep`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, float]

See [`ARKodeGetCurrentStep\(\)`](#).

`sundials4py.arkode.ARKodeGetCurrentTime`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, float]

See [`ARKodeGetCurrentTime\(\)`](#).

`sundials4py.arkode.ARKodeGetDky`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *t*: float, *k*: int, *dky*: *sundials4py.core.\_generic\_N\_Vector*) → int

Computes the kth derivative of the y function at time t

See [`ARKodeGetDky\(\)`](#).

`sundials4py.arkode.ARKodeGetErrWeights`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *eweight*: *sundials4py.core.\_generic\_N\_Vector*) → int

See [`ARKodeGetErrWeights\(\)`](#).

`sundials4py.arkode.ARKodeGetEstLocalErrors`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *ele*: *sundials4py.core.\_generic\_N\_Vector*) → int

See [`ARKodeGetEstLocalErrors\(\)`](#).

`sundials4py.arkode.ARKodeGetJac`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, *sundials4py.core.\_generic\_SUNMatrix*]

nb::rv\_policy::reference

See [`ARKodeGetJac\(\)`](#).

`sundials4py.arkode.ARKodeGetJacNumSteps`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, int]

See [`ARKodeGetJacNumSteps\(\)`](#).

`sundials4py.arkode.ARKodeGetJacTime`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, float]

See [`ARKodeGetJacTime\(\)`](#).

`sundials4py.arkode.ARKodeGetLastLinFlag`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, int]

See [`ARKodeGetLastLinFlag\(\)`](#).

`sundials4py.arkode.ARKodeGetLastMassFlag`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, int]

See [`ARKodeGetLastMassFlag\(\)`](#).

`sundials4py.arkode.ARKodeGetLastStep`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, float]

See [`ARKodeGetLastStep\(\)`](#).

`sundials4py.arkode.ARKodeGetLinReturnFlagName`(*flag*: int) → str

See [`ARKodeGetLinReturnFlagName\(\)`](#).

`sundials4py.arkode.ARKodeGetNonlinSolvStats`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, int, int]

See [`ARKodeGetNonlinSolvStats\(\)`](#).

`sundials4py.arkode.ARKodeGetNumAccSteps`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, int]

See [`ARKodeGetNumAccSteps\(\)`](#).

`sundials4py.arkode.ARKodeGetNumConstrFails`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → tuple[int, int]

See [`ARKodeGetNumConstrFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumErrTestFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumErrTestFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumExpSteps(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumExpSteps\(\)`](#).

`sundials4py.arkode.ARKodeGetNumGEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumGEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumJTSetupEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumJTSetupEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumJacEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumJacEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumJtimesEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumJtimesEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumLinConvFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumLinConvFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumLinIters(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumLinIters\(\)`](#).

`sundials4py.arkode.ARKodeGetNumLinRhsEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumLinRhsEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumLinSolvSetups(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumLinSolvSetups\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMTSetups(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMTSetups\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassConvFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassConvFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassIters(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassIters\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassMult(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassMult\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassMultSetups(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassMultSetups\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassPrecEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassPrecEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassPrecSolves(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassPrecSolves\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassSetups(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassSetups\(\)`](#).

`sundials4py.arkode.ARKodeGetNumMassSolves(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumMassSolves\(\)`](#).

`sundials4py.arkode.ARKodeGetNumNonlinSolvConvFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumNonlinSolvConvFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumNonlinSolvIters(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumNonlinSolvIters\(\)`](#).

`sundials4py.arkode.ARKodeGetNumPrecEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumPrecEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumPrecSolves(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumPrecSolves\(\)`](#).

`sundials4py.arkode.ARKodeGetNumRelaxBoundFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumRelaxBoundFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumRelaxFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumRelaxFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumRelaxFnEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumRelaxFnEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumRelaxJacEvals(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumRelaxJacEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumRelaxSolveFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumRelaxSolveFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumRelaxSolveIters(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumRelaxSolveIters\(\)`](#).

`sundials4py.arkode.ARKodeGetNumRhsEvals(arkode_mem: typing_extensions.CapsuleType, partition_index: int) → tuple[int, int]`

See [`ARKodeGetNumRhsEvals\(\)`](#).

`sundials4py.arkode.ARKodeGetNumStepAttempts(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumStepAttempts\(\)`](#).

`sundials4py.arkode.ARKodeGetNumStepSolveFails(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumStepSolveFails\(\)`](#).

`sundials4py.arkode.ARKodeGetNumSteps(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int]`

See [`ARKodeGetNumSteps\(\)`](#).

`sundials4py.arkode.ARKodeGetResWeights(arkode_mem: typing_extensions.CapsuleType, rweight: sundials4py.core._generic_N_Vector) → int`

See [`ARKodeGetResWeights\(\)`](#).

`sundials4py.arkode.ARKodeGetReturnFlagName(flag: int) → str`

See [`ARKodeGetReturnFlagName\(\)`](#).

`sundials4py.arkode.ARKodeGetRootInfo(arkode_mem: typing_extensions.CapsuleType, rootsfound_id: collections.abc.Sequence[int]) → int`

See [`ARKodeGetRootInfo\(\)`](#).

`sundials4py.arkode.ARKodeGetStepDirection(arkode_mem: typing_extensions.CapsuleType) → tuple[int, float]`

See [`ARKodeGetStepDirection\(\)`](#).

`sundials4py.arkode.ARKodeGetStepStats(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int, float, float, float, float]`

See [`ARKodeGetStepStats\(\)`](#).

`sundials4py.arkode.ARKodeGetTolScaleFactor(arkode_mem: typing_extensions.CapsuleType) → tuple[int, float]`

See [`ARKodeGetTolScaleFactor\(\)`](#).

`sundials4py.arkode.ARKodePrintAllStats(arkode_mem: typing_extensions.CapsuleType, outfile: sundials4py.core.FILE, fmt: sundials4py.core.SUNOutputFormat) → int`

See [`ARKodePrintAllStats\(\)`](#).

`sundials4py.arkode.ARKodePrintMem(arkode_mem: typing_extensions.CapsuleType, outfile: sundials4py.core.FILE) → None`

Output the ARKODE memory structure (useful when debugging)

See [`ARKodePrintMem\(\)`](#).

`sundials4py.arkode.ARKodeResFtolerance(arkode_mem: typing_extensions.CapsuleType, efun: collections.abc.Callable[[sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`

See [`ARKodeResFtolerance\(\)`](#).

`sundials4py.arkode.ARKodeResStolerance(arkode_mem: typing_extensions.CapsuleType, rabstol: float) → int`

See [`ARKodeResStolerance\(\)`](#).

`sundials4py.arkode.ARKodeResVtolerance(arkode_mem: typing_extensions.CapsuleType, rabstol: sundials4py.core._generic_N_Vector) → int`

See [`ARKodeResVtolerance\(\)`](#).

`sundials4py.arkode.ARKodeReset(arkode_mem: typing_extensions.CapsuleType, tR: float, yR: sundials4py.core._generic_N_Vector) → int`

See [`ARKodeReset\(\)`](#).

`sundials4py.arkode.ARKodeResetAccumulatedError(arkode_mem: typing_extensions.CapsuleType) → int`

See [`ARKodeResetAccumulatedError\(\)`](#).

`sundials4py.arkode.ARKodeResize(arkode_mem: typing_extensions.CapsuleType, y_new: sundials4py.core._generic_N_Vector, h_scale: float, t0: float, resize_fn: collections.abc.Callable[[sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`

See [`ARKodeResize\(\)`](#).

`sundials4py.arkode.ARKodeRootInit(arkode_mem: typing_extensions.CapsuleType, nrtfn: int, root_fn: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, numpy.ndarray[dtype=float64, shape=(*), order='C'], typing_extensions.CapsuleType], int] | None) → int`

See [`ARKodeRootInit\(\)`](#).

`sundials4py.arkode.ARKodeSPRKTable_Copy(that_sprk_storage: sundials4py.arkode.ARKodeSPRKTableMem) → sundials4py.arkode.ARKodeSPRKTableMem`

See [`ARKodeSPRKTable\_Copy\(\)`](#).

`sundials4py.arkode.ARKodeSPRKTable_Create(s: int, q: int, a_1d: numpy.ndarray[dtype=float64, shape=(*), order='C'], ahat_1d: numpy.ndarray[dtype=float64, shape=(*), order='C']) → sundials4py.arkode.ARKodeSPRKTableMem`

See [`ARKodeSPRKTable\_Create\(\)`](#).

`sundials4py.arkode.ARKodeSPRKTable_Load(id: sundials4py.arkode.ARKODE_SPRKMethodID) → sundials4py.arkode.ARKodeSPRKTableMem`

See [`ARKodeSPRKTable\_Load\(\)`](#).

`sundials4py.arkode.ARKodeSPRKTable_LoadByName(method: str) → sundials4py.arkode.ARKodeSPRKTableMem`

See [`ARKodeSPRKTable\_LoadByName\(\)`](#).

`sundials4py.arkode.ARKodeSPRKTable_ToButcher(sprk_storage: sundials4py.arkode.ARKodeSPRKTableMem) → tuple[int, sundials4py.arkode.ARKodeButcherTableMem, sundials4py.arkode.ARKodeButcherTableMem]`

See [`ARKodeSPRKTable\_ToButcher\(\)`](#).

`sundials4py.arkode.ARKodeSPRKTable_Write(sprk_table: sundials4py.arkode.ARKodeSPRKTableMem, outfile: sundials4py.core.FILE) → None`

See [`ARKodeSPRKTable\_Write\(\)`](#).

`sundials4py.arkode.ARKodeSStolerances(arkode_mem: typing_extensions.CapsuleType, reltol: float, abstol: float) → int`

See [`ARKodeSStolerances\(\)`](#).

`sundials4py.arkode.ARKodeSVtolerances(arkode_mem: typing_extensions.CapsuleType, reltol: float, abstol: sundials4py.core._generic_N_Vector) → int`

See [`ARKodeSVtolerances\(\)`](#).

`sundials4py.arkode.ARKodeSetAccumulatedErrorType(arkode_mem: typing_extensions.CapsuleType, accum_type: sundials4py.arkode.ARKAccumError) → int`

See [`ARKodeSetAccumulatedErrorType\(\)`](#).

`sundials4py.arkode.ARKodeSetAdaptController(arkode_mem: typing_extensions.CapsuleType, C: sundials4py.core._generic_SUNAdaptController) → int`

See [`ARKodeSetAdaptController\(\)`](#).

`sundials4py.arkode.ARKodeSetAdaptControllerByName(arkode_mem: typing_extensions.CapsuleType, cname: str) → int`

See [`ARKodeSetAdaptControllerByName\(\)`](#).

`sundials4py.arkode.ARKodeSetAdaptivityAdjustment(arkode_mem: typing_extensions.CapsuleType, adjust: int) → int`

See [`ARKodeSetAdaptivityAdjustment\(\)`](#).

`sundials4py.arkode.ARKodeSetAdjointCheckpointIndex(arkode_mem: typing_extensions.CapsuleType, step_index: int) → int`

See [`ARKodeSetAdjointCheckpointIndex\(\)`](#).

`sundials4py.arkode.ARKodeSetAdjointCheckpointScheme(arkode_mem: typing_extensions.CapsuleType, checkpoint_scheme: sundials4py.core.SUNAdjointCheckpointScheme_) → int`

See [`ARKodeSetAdjointCheckpointScheme\(\)`](#).

`sundials4py.arkode.ARKodeSetAutonomous(arkode_mem: typing_extensions.CapsuleType, autonomous: int) → int`

See [`ARKodeSetAutonomous\(\)`](#).

`sundials4py.arkode.ARKodeSetCFLFraction(arkode_mem: typing_extensions.CapsuleType, cfl_frac: float) → int`

See [`ARKodeSetCFLFraction\(\)`](#).

`sundials4py.arkode.ARKodeSetConstraints(arkode_mem: typing_extensions.CapsuleType, constraints: sundials4py.core._generic_N_Vector) → int`

See [`ARKodeSetConstraints\(\)`](#).

`sundials4py.arkode.ARKodeSetDeduceImplicitRhs(arkode_mem: typing_extensions.CapsuleType, deduce: int) → int`

See [`ARKodeSetDeduceImplicitRhs\(\)`](#).

`sundials4py.arkode.ARKodeSetDefaults(arkode_mem: typing_extensions.CapsuleType) → int`

See [`ARKodeSetDefaults\(\)`](#).

`sundials4py.arkode.ARKodeSetDeltaGammaMax(arkode_mem: typing_extensions.CapsuleType, dgmax: float) → int`

See [`ARKodeSetDeltaGammaMax\(\)`](#).

`sundials4py.arkode.ARKodeSetEpsLin(arkode_mem: typing_extensions.CapsuleType, eplifac: float) → int`

See [`ARKodeSetEpsLin\(\)`](#).



`sundials4py.arkode.ARKodeSetErrorBias(arkode_mem: typing_extensions.CapsuleType, bias: float) → int`  
 See [ARKodeSetErrorBias\(\)](#).

`sundials4py.arkode.ARKodeSetFixedStep(arkode_mem: typing_extensions.CapsuleType, hfixed: float) → int`  
 See [ARKodeSetFixedStep\(\)](#).

`sundials4py.arkode.ARKodeSetFixedStepBounds(arkode_mem: typing_extensions.CapsuleType, lb: float, ub: float) → int`  
 See [ARKodeSetFixedStepBounds\(\)](#).

`sundials4py.arkode.ARKodeSetInitStep(arkode_mem: typing_extensions.CapsuleType, hin: float) → int`  
 See [ARKodeSetInitStep\(\)](#).

`sundials4py.arkode.ARKodeSetInterpolantDegree(arkode_mem: typing_extensions.CapsuleType, degree: int) → int`  
 See [ARKodeSetInterpolantDegree\(\)](#).

`sundials4py.arkode.ARKodeSetInterpolantType(arkode_mem: typing_extensions.CapsuleType, itype: int) → int`  
 See [ARKodeSetInterpolantType\(\)](#).

`sundials4py.arkode.ARKodeSetInterpolateStopTime(arkode_mem: typing_extensions.CapsuleType, interp: int) → int`  
 See [ARKodeSetInterpolateStopTime\(\)](#).

`sundials4py.arkode.ARKodeSetJacEvalFrequency(arkode_mem: typing_extensions.CapsuleType, msbj: int) → int`  
 See [ARKodeSetJacEvalFrequency\(\)](#).

`sundials4py.arkode.ARKodeSetJacFn(arkode_mem: typing_extensions.CapsuleType, jac: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core._generic_SUNMatrix, typing_extensions.CapsuleType, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector], int] | None) → int`  
 See [ARKodeSetJacFn\(\)](#).

`sundials4py.arkode.ARKodeSetJacTimes(arkode_mem: typing_extensions.CapsuleType, jtsetup: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None, jt看times: collections.abc.Callable[[sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType, sundials4py.core._generic_N_Vector], int] | None) → int`  
 See [ARKodeSetJacTimes\(\)](#).

`sundials4py.arkode.ARKodeSetJacTimesRhsFn(arkode_mem: typing_extensions.CapsuleType, jt看timesRhsFn: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`

See [ARKodeSetJacTimesRhsFn\(\)](#).

`sundials4py.arkode.ARKodeSetLSNormFactor(arkode_mem: typing_extensions.CapsuleType, nrmfac: float) → int`

See [ARKodeSetLSNormFactor\(\)](#).

`sundials4py.arkode.ARKodeSetLSetupFrequency(arkode_mem: typing_extensions.CapsuleType, msbp: int) → int`

See [ARKodeSetLSetupFrequency\(\)](#).

`sundials4py.arkode.ARKodeSetLinSysFn(arkode_mem: typing_extensions.CapsuleType, linsys: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core._generic_SUNMatrix, sundials4py.core._generic_SUNMatrix, int, int, float, typing_extensions.CapsuleType, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector], int] | None) → int`

See [ARKodeSetLinSysFn\(\)](#).

`sundials4py.arkode.ARKodeSetLinear(arkode_mem: typing_extensions.CapsuleType, timedepend: int) → int`

See [ARKodeSetLinear\(\)](#).

`sundials4py.arkode.ARKodeSetLinearSolutionScaling(arkode_mem: typing_extensions.CapsuleType, onoff: int) → int`

See [ARKodeSetLinearSolutionScaling\(\)](#).

`sundials4py.arkode.ARKodeSetLinearSolver(arkode_mem: typing_extensions.CapsuleType, LS: sundials4py.core._generic_SUNLinearSolver, A: sundials4py.core._generic_SUNMatrix | None = None) → int`

See [ARKodeSetLinearSolver\(\)](#).

`sundials4py.arkode.ARKodeSetMassEpsLin(arkode_mem: typing_extensions.CapsuleType, eplifac: float) → int`

See [ARKodeSetMassEpsLin\(\)](#).

`sundials4py.arkode.ARKodeSetMassFn(arkode_mem: typing_extensions.CapsuleType, mass: collections.abc.Callable[[float, sundials4py.core._generic_SUNMatrix, typing_extensions.CapsuleType, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector], int] | None) → int`

See [ARKodeSetMassFn\(\)](#).

`sundials4py.arkode.ARKodeSetMassLSNormFactor(arkode_mem: typing_extensions.CapsuleType, nrmfac: float) → int`

See [ARKodeSetMassLSNormFactor\(\)](#).

`sundials4py.arkode.ARKodeSetMassLinearSolver(arkode_mem: typing_extensions.CapsuleType, LS: sundials4py.core._generic_SUNLinearSolver, M: sundials4py.core._generic_SUNMatrix | None, time_dep: int) → int`

See [ARKodeSetMassLinearSolver\(\)](#).



`sundials4py.arkode.ARKodeSetMassPreconditioner`(*arkode\_mem*: `typing_extensions.CapsuleType`, *psetup*: `collections.abc.Callable[[float, typing_extensions.CapsuleType], int] | None`, *psolve*: `collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, float, int, typing_extensions.CapsuleType], int] | None`) → `int`

See `ARKodeSetMassPreconditioner()`.

`sundials4py.arkode.ARKodeSetMassTimes`(*ark\_mem*: `typing_extensions.CapsuleType`, *msetup*: `collections.abc.Callable[[float, typing_extensions.CapsuleType], int] | None`, *mtimes*: `collections.abc.Callable[[sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, float, typing_extensions.CapsuleType], int] | None`) → `int`

See `ARKodeSetMassTimes()`.

`sundials4py.arkode.ARKodeSetMaxCFailGrowth`(*arkode\_mem*: `typing_extensions.CapsuleType`, *etacf*: `float`) → `int`

See `ARKodeSetMaxCFailGrowth()`.

`sundials4py.arkode.ARKodeSetMaxConvFails`(*arkode\_mem*: `typing_extensions.CapsuleType`, *maxncf*: `int`) → `int`

See `ARKodeSetMaxConvFails()`.

`sundials4py.arkode.ARKodeSetMaxEFailGrowth`(*arkode\_mem*: `typing_extensions.CapsuleType`, *etamxf*: `float`) → `int`

See `ARKodeSetMaxEFailGrowth()`.

`sundials4py.arkode.ARKodeSetMaxErrTestFails`(*arkode\_mem*: `typing_extensions.CapsuleType`, *maxnef*: `int`) → `int`

See `ARKodeSetMaxErrTestFails()`.

`sundials4py.arkode.ARKodeSetMaxFirstGrowth`(*arkode\_mem*: `typing_extensions.CapsuleType`, *etamx1*: `float`) → `int`

See `ARKodeSetMaxFirstGrowth()`.

`sundials4py.arkode.ARKodeSetMaxGrowth`(*arkode\_mem*: `typing_extensions.CapsuleType`, *mx\_growth*: `float`) → `int`

See `ARKodeSetMaxGrowth()`.

`sundials4py.arkode.ARKodeSetMaxHnilWarns`(*arkode\_mem*: `typing_extensions.CapsuleType`, *mxhnil*: `int`) → `int`

See `ARKodeSetMaxHnilWarns()`.

`sundials4py.arkode.ARKodeSetMaxNonlinIters`(*arkode\_mem*: `typing_extensions.CapsuleType`, *maxcor*: `int`) → `int`

See `ARKodeSetMaxNonlinIters()`.

`sundials4py.arkode.ARKodeSetMaxNumConstrFails`(*arkode\_mem*: `typing_extensions.CapsuleType`, *maxfails*: `int`) → `int`

See `ARKodeSetMaxNumConstrFails()`.

`sundials4py.arkode.ARKodeSetMaxNumSteps`(*arkode\_mem*: `typing_extensions.CapsuleType`, *mxsteps*: `int`) → `int`

See `ARKodeSetMaxNumSteps()`.

`sundials4py.arkode.ARKodeSetMaxStep(arkode_mem: typing_extensions.CapsuleType, hmax: float) → int`  
See [`ARKodeSetMaxStep\(\)`](#).

`sundials4py.arkode.ARKodeSetMinReduction(arkode_mem: typing_extensions.CapsuleType, eta_min: float) → int`  
See [`ARKodeSetMinReduction\(\)`](#).

`sundials4py.arkode.ARKodeSetMinStep(arkode_mem: typing_extensions.CapsuleType, hmin: float) → int`  
See [`ARKodeSetMinStep\(\)`](#).

`sundials4py.arkode.ARKodeSetNlsRhsFn(arkode_mem: typing_extensions.CapsuleType, nls_fn: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`  
See [`ARKodeSetNlsRhsFn\(\)`](#).

`sundials4py.arkode.ARKodeSetNoInactiveRootWarn(arkode_mem: typing_extensions.CapsuleType) → int`  
See [`ARKodeSetNoInactiveRootWarn\(\)`](#).

`sundials4py.arkode.ARKodeSetNonlinCRDown(arkode_mem: typing_extensions.CapsuleType, crdown: float) → int`  
See [`ARKodeSetNonlinCRDown\(\)`](#).

`sundials4py.arkode.ARKodeSetNonlinConvCoef(arkode_mem: typing_extensions.CapsuleType, nlscoef: float) → int`  
See [`ARKodeSetNonlinConvCoef\(\)`](#).

`sundials4py.arkode.ARKodeSetNonlinRDiv(arkode_mem: typing_extensions.CapsuleType, rdiv: float) → int`  
See [`ARKodeSetNonlinRDiv\(\)`](#).

`sundials4py.arkode.ARKodeSetNonlinear(arkode_mem: typing_extensions.CapsuleType) → int`  
See [`ARKodeSetNonlinear\(\)`](#).

`sundials4py.arkode.ARKodeSetNonlinearSolver(arkode_mem: typing_extensions.CapsuleType, NLS: sundials4py.core._generic_SUNNonlinearSolver) → int`  
See [`ARKodeSetNonlinearSolver\(\)`](#).

`sundials4py.arkode.ARKodeSetOptions(ark_mem: typing_extensions.CapsuleType, arkid: str, file_name: str, argc: int, args: collections.abc.Sequence[str]) → int`  
See [`ARKodeSetOptions\(\)`](#).

`sundials4py.arkode.ARKodeSetOrder(arkode_mem: typing_extensions.CapsuleType, maxord: int) → int`  
See [`ARKodeSetOrder\(\)`](#).

`sundials4py.arkode.ARKodeSetPostprocessStageFn(arkode_mem: typing_extensions.CapsuleType, postprocessstage: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`  
See [`ARKodeSetPostprocessStageFn\(\)`](#).

`sundials4py.arkode.ARKodeSetPostprocessStepFn(arkode_mem: typing_extensions.CapsuleType, postprocessstep: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`  
See [`ARKodeSetPostprocessStepFn\(\)`](#).

`sundials4py.arkode.ARKodeSetPreconditioner`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *psetup*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *int*, *int*, *float*, *typing\_extensions.CapsuleType*], *int*] | *None*, *psolve*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *float*, *float*, *int*, *typing\_extensions.CapsuleType*], *int*] | *None*) → *int*

See [`ARKodeSetPreconditioner\(\)`](#).

`sundials4py.arkode.ARKodeSetPredictorMethod`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *method*: *int*) → *int*

See [`ARKodeSetPredictorMethod\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxEtaFail`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *eta\_rf*: *float*) → *int*

See [`ARKodeSetRelaxEtaFail\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxFn`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *rfn*: *collections.abc.Callable*[[*sundials4py.core.\_generic\_N\_Vector*, *float*, *typing\_extensions.CapsuleType*], *int*] | *None*, *rjacfn*: *collections.abc.Callable*[[*sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*] | *None*) → *int*

See [`ARKodeSetRelaxFn\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxLowerBound`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *lower*: *float*) → *int*

See [`ARKodeSetRelaxLowerBound\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxMaxFails`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *max\_fails*: *int*) → *int*

See [`ARKodeSetRelaxMaxFails\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxMaxIters`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *max\_iters*: *int*) → *int*

See [`ARKodeSetRelaxMaxIters\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxResTol`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *res\_tol*: *float*) → *int*

See [`ARKodeSetRelaxResTol\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxSolver`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *solver*: *sundials4py.arkode.ARKRelaxSolver*) → *int*

See [`ARKodeSetRelaxSolver\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxTol`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *rel\_tol*: *float*, *abs\_tol*: *float*) → *int*

See [`ARKodeSetRelaxTol\(\)`](#).

`sundials4py.arkode.ARKodeSetRelaxUpperBound`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *upper*: *float*) → *int*

See [`ARKodeSetRelaxUpperBound\(\)`](#).

`sundials4py.arkode.ARKodeSetRootDirection(arkode_mem: typing_extensions.CapsuleType, rootdir_id: collections.abc.Sequence[int]) → int`

See [`ARKodeSetRootDirection\(\)`](#).

`sundials4py.arkode.ARKodeSetSafetyFactor(arkode_mem: typing_extensions.CapsuleType, safety: float) → int`

See [`ARKodeSetSafetyFactor\(\)`](#).

`sundials4py.arkode.ARKodeSetSmallNumEFails(arkode_mem: typing_extensions.CapsuleType, small_nef: int) → int`

See [`ARKodeSetSmallNumEFails\(\)`](#).

`sundials4py.arkode.ARKodeSetStagePredictFn(arkode_mem: typing_extensions.CapsuleType, stagepredict: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`

See [`ARKodeSetStagePredictFn\(\)`](#).

`sundials4py.arkode.ARKodeSetStepDirection(arkode_mem: typing_extensions.CapsuleType, stepdir: float) → int`

See [`ARKodeSetStepDirection\(\)`](#).

`sundials4py.arkode.ARKodeSetStopTime(arkode_mem: typing_extensions.CapsuleType, tstop: float) → int`

See [`ARKodeSetStopTime\(\)`](#).

`sundials4py.arkode.ARKodeSetUseCompensatedSums(arkode_mem: typing_extensions.CapsuleType, onoff: int) → int`

See [`ARKodeSetUseCompensatedSums\(\)`](#).

`sundials4py.arkode.ARKodeWFTolerances(arkode_mem: typing_extensions.CapsuleType, efun: collections.abc.Callable[[sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None) → int`

See [`ARKodeWFTolerances\(\)`](#).

`sundials4py.arkode.ARKodeWriteParameters(arkode_mem: typing_extensions.CapsuleType, fp: sundials4py.core.FILE) → int`

See [`ARKodeWriteParameters\(\)`](#).

`sundials4py.arkode.ERKStepCreate(rhs: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int], t0: float, y0: sundials4py.core._generic_N_Vector, sunctx: sundials4py.core.SUNContext_) → sundials4py.arkode.ARKodeView`

See [`ERKStepCreate\(\)`](#).

`sundials4py.arkode.ERKStepCreateAdjointStepper(arkode_mem: typing_extensions.CapsuleType, adj_f: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int] | None, tf: float, sf: sundials4py.core._generic_N_Vector, sunctx: sundials4py.core.SUNContext_) → tuple[int, sundials4py.core.SUNAdjointStepper_]`

See [`ERKStepCreateAdjointStepper\(\)`](#).

`sundials4py.arkode.ERKStepGetCurrentButcherTable(arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.arkode.ARKodeButcherTableMem]`

Optional output functions

`nb::rv_policy::reference`

See [`ERKStepGetCurrentButcherTable\(\)`](#).

`sundials4py.arkode.ERKStepGetTimestepperStats(arkode_mem: typing_extensions.CapsuleType) → tuple[int, int, int, int, int, int]`

Grouped optional output functions

See [`ERKStepGetTimestepperStats\(\)`](#).

`sundials4py.arkode.ERKStepSetTable(arkode_mem: typing_extensions.CapsuleType, B: sundials4py.arkode.ARKodeButcherTableMem) → int`

See [`ERKStepSetTable\(\)`](#).

`sundials4py.arkode.ERKStepSetTableName(arkode_mem: typing_extensions.CapsuleType, etable: str) → int`  
See [`ERKStepSetTableName\(\)`](#).

`sundials4py.arkode.ERKStepSetTableNum(arkode_mem: typing_extensions.CapsuleType, etable: sundials4py.arkode.ARKODE_ERKTableID) → int`

See [`ERKStepSetTableNum\(\)`](#).

`sundials4py.arkode.ForcingStepCreate(stepper1: sundials4py.core.SUNStepper_, stepper2: sundials4py.core.SUNStepper_, t0: float, y0: sundials4py.core._generic_N_Vector, sunctx: sundials4py.core.SUNContext_) → sundials4py.arkode.ARKodeView`

See [`ForcingStepCreate\(\)`](#).

`sundials4py.arkode.ForcingStepGetNumEvolves(arkode_mem: typing_extensions.CapsuleType, partition: int) → tuple[int, int]`

See [`ForcingStepGetNumEvolves\(\)`](#).

`sundials4py.arkode.ForcingStepReInit(arkode_mem: typing_extensions.CapsuleType, stepper1: sundials4py.core.SUNStepper_, stepper2: sundials4py.core.SUNStepper_, t0: float, y0: sundials4py.core._generic_N_Vector) → int`

See [`ForcingStepReInit\(\)`](#).

`sundials4py.arkode.LSRKStepCreateSSP(rhs: collections.abc.Callable[[float, sundials4py.core._generic_N_Vector, sundials4py.core._generic_N_Vector, typing_extensions.CapsuleType], int], t0: float, y0: sundials4py.core._generic_N_Vector, sunctx: sundials4py.core.SUNContext_) → sundials4py.arkode.ARKodeView`

See [`LSRKStepCreateSSP\(\)`](#).

`sundials4py.arkode.LSRKStepCreateSTS`(*rhs*: *collections.abc.Callable*[[*float*,  
*sundials4py.core.\_generic\_N\_Vector*,  
*sundials4py.core.\_generic\_N\_Vector*,  
*typing\_extensions.CapsuleType*], *int*], *t0*: *float*, *y0*:  
*sundials4py.core.\_generic\_N\_Vector*, *sunctx*:  
*sundials4py.core.SUNContext\_*) →  
*sundials4py.arkode.ARKodeView*

See [`LSRKStepCreateSTS\(\)`](#).

`sundials4py.arkode.LSRKStepGetMaxNumStages`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → *tuple*[*int*,  
*int*]

See [`LSRKStepGetMaxNumStages\(\)`](#).

`sundials4py.arkode.LSRKStepGetNumDomEigEstIters`(*arkode\_mem*: *typing\_extensions.CapsuleType*) →  
*tuple*[*int*, *int*]

See [`LSRKStepGetNumDomEigEstIters\(\)`](#).

`sundials4py.arkode.LSRKStepGetNumDomEigEstRhsEvals`(*arkode\_mem*: *typing\_extensions.CapsuleType*) →  
*tuple*[*int*, *int*]

See [`LSRKStepGetNumDomEigEstRhsEvals\(\)`](#).

`sundials4py.arkode.LSRKStepGetNumDomEigUpdates`(*arkode\_mem*: *typing\_extensions.CapsuleType*) →  
*tuple*[*int*, *int*]

See [`LSRKStepGetNumDomEigUpdates\(\)`](#).

`sundials4py.arkode.LSRKStepSetDomEigEstimator`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *DEE*:  
*sundials4py.core.SUNDomEigEstimator\_*) → *int*

See [`LSRKStepSetDomEigEstimator\(\)`](#).

`sundials4py.arkode.LSRKStepSetDomEigFn`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *eig\_fn*:  
*collections.abc.Callable*[[*float*,  
*sundials4py.core.\_generic\_N\_Vector*,  
*sundials4py.core.\_generic\_N\_Vector*, *float*, *float*,  
*typing\_extensions.CapsuleType*,  
*sundials4py.core.\_generic\_N\_Vector*,  
*sundials4py.core.\_generic\_N\_Vector*,  
*sundials4py.core.\_generic\_N\_Vector*], *int*]) → *int*

See [`LSRKStepSetDomEigFn\(\)`](#).

`sundials4py.arkode.LSRKStepSetDomEigFrequency`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *nsteps*:  
*int*) → *int*

See [`LSRKStepSetDomEigFrequency\(\)`](#).

`sundials4py.arkode.LSRKStepSetDomEigSafetyFactor`(*arkode\_mem*: *typing\_extensions.CapsuleType*,  
*dom\_eig\_safety*: *float*) → *int*

See [`LSRKStepSetDomEigSafetyFactor\(\)`](#).

`sundials4py.arkode.LSRKStepSetMaxNumStages`(*arkode\_mem*: *typing\_extensions.CapsuleType*,  
*stage\_max\_limit*: *int*) → *int*

See [`LSRKStepSetMaxNumStages\(\)`](#).

`sundials4py.arkode.LSRKStepSetNumDomEigEstInitPreprocessIters`(*arkode\_mem*:  
*typing\_extensions.CapsuleType*,  
*num\_iters*: *int*) → *int*

See [`LSRKStepSetNumDomEigEstInitPreprocessIters\(\)`](#).



`sundials4py.arkode.LSRKStepSetNumDomEigEstPreprocessIters`(*arkode\_mem*:  
*typing\_extensions.CapsuleType*,  
*num\_iters*: *int*) → *int*

See [`LSRKStepSetNumDomEigEstPreprocessIters\(\)`](#).

`sundials4py.arkode.LSRKStepSetNumSSPStages`(*arkode\_mem*: *typing\_extensions.CapsuleType*,  
*num\_of\_stages*: *int*) → *int*

See [`LSRKStepSetNumSSPStages\(\)`](#).

`sundials4py.arkode.LSRKStepSetSSPMethod`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *method*:  
*sundials4py.arkode.ARKODE\_LSRKMethodType*) → *int*

See [`LSRKStepSetSSPMethod\(\)`](#).

`sundials4py.arkode.LSRKStepSetSSPMethodByName`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *emethod*:  
*str*) → *int*

See [`LSRKStepSetSSPMethodByName\(\)`](#).

`sundials4py.arkode.LSRKStepSetSTSMethod`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *method*:  
*sundials4py.arkode.ARKODE\_LSRKMethodType*) → *int*

See [`LSRKStepSetSTSMethod\(\)`](#).

`sundials4py.arkode.LSRKStepSetSTSMethodByName`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *emethod*:  
*str*) → *int*

See [`LSRKStepSetSTSMethodByName\(\)`](#).

`sundials4py.arkode.MRIStepCoupling_Copy`(*MRIC*: *sundials4py.arkode.MRIStepCouplingMem*) →  
*sundials4py.arkode.MRIStepCouplingMem*

See [`MRIStepCoupling\_Copy\(\)`](#).

`sundials4py.arkode.MRIStepCoupling_Create`(*nmat*: *int*, *stages*: *int*, *q*: *int*, *p*: *int*, *W\_1d*:  
*numpy.ndarray*[*dtype*=*float64*, *shape*=(*\**), *order*=*'C'*], *G\_1d*:  
*numpy.ndarray*[*dtype*=*float64*, *shape*=(*\**), *order*=*'C'*], *c\_1d*:  
*numpy.ndarray*[*dtype*=*float64*, *shape*=(*\**), *order*=*'C'*]) →  
*sundials4py.arkode.MRIStepCouplingMem*

See [`MRIStepCoupling\_Create\(\)`](#).

`sundials4py.arkode.MRIStepCoupling_LoadTable`(*method*: *sundials4py.arkode.ARKODE\_MRITableID*) →  
*sundials4py.arkode.MRIStepCouplingMem*

Accessor routine to load built-in MRI table

See [`MRIStepCoupling\_LoadTable\(\)`](#).

`sundials4py.arkode.MRIStepCoupling_LoadTableByName`(*method*: *str*) →  
*sundials4py.arkode.MRIStepCouplingMem*

Accessor routine to load built-in MRI table from string

See [`MRIStepCoupling\_LoadTableByName\(\)`](#).

`sundials4py.arkode.MRIStepCoupling_MISToMRI`(*B*: *sundials4py.arkode.ARKodeButcherTableMem*, *q*: *int*,  
*p*: *int*) → *sundials4py.arkode.MRIStepCouplingMem*

See [`MRIStepCoupling\_MISToMRI\(\)`](#).

`sundials4py.arkode.MRIStepCoupling_Write`(*MRIC*: *sundials4py.arkode.MRIStepCouplingMem*, *outfile*:  
*sundials4py.core.FILE*) → *None*

See [`MRIStepCoupling\_Write\(\)`](#).

`sundials4py.arkode.MRIStepCreate`(*fse*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*] | *None*, *fsi*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*] | *None*, *t0*: *float*, *y0*: *sundials4py.core.\_generic\_N\_Vector*, *inner\_stepper*: *sundials4py.arkode.\_MRIStepInnerStepper*, *sunctx*: *sundials4py.core.SUNContext\_*) → *sundials4py.arkode.ARKodeView*

See [`MRIStepCreate\(\)`](#).

`sundials4py.arkode.MRIStepGetCurrentCoupling`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → *tuple*[*int*, *sundials4py.arkode.MRIStepCouplingMem*]

Optional output functions

`nb::rv_policy::reference`

See [`MRIStepGetCurrentCoupling\(\)`](#).

`sundials4py.arkode.MRIStepGetLastInnerStepFlag`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → *tuple*[*int*, *int*]

See [`MRIStepGetLastInnerStepFlag\(\)`](#).

`sundials4py.arkode.MRIStepGetNumInnerStepperFails`(*arkode\_mem*: *typing\_extensions.CapsuleType*) → *tuple*[*int*, *int*]

See [`MRIStepGetNumInnerStepperFails\(\)`](#).

`sundials4py.arkode.MRIStepInnerStepper_AddForcing`(*stepper*: *sundials4py.arkode.\_MRIStepInnerStepper*, *t*: *float*, *f*: *sundials4py.core.\_generic\_N\_Vector*) → *int*

See [`MRIStepInnerStepper\_AddForcing\(\)`](#).

`sundials4py.arkode.MRIStepInnerStepper_Create`(*sunctx*: *sundials4py.core.SUNContext\_*) → *tuple*[*int*, *sundials4py.arkode.\_MRIStepInnerStepper*]

See [`MRIStepInnerStepper\_Create\(\)`](#).

`sundials4py.arkode.MRIStepInnerStepper_CreateFromSUNStepper`(*stepper*: *sundials4py.core.SUNStepper\_*) → *tuple*[*int*, *sundials4py.arkode.\_MRIStepInnerStepper*]

See [`MRIStepInnerStepper\_CreateFromSUNStepper\(\)`](#).

`sundials4py.arkode.MRIStepInnerStepper_GetForcingData`(*arg*: *sundials4py.arkode.\_MRIStepInnerStepper*, */*) → *tuple*[*int*, *float*, *float*, *list*[*sundials4py.core.\_generic\_N\_Vector*], *int*]

See [`MRIStepInnerStepper\_GetForcingData\(\)`](#).

`sundials4py.arkode.MRIStepSetCoupling`(*arkode\_mem*: *typing\_extensions.CapsuleType*, *MRIC*: *sundials4py.arkode.MRIStepCouplingMem*) → *int*

See [`MRIStepSetCoupling\(\)`](#).

`sundials4py.arkode.SPRKStepCreate`(*f1*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*], *f2*: *collections.abc.Callable*[[*float*, *sundials4py.core.\_generic\_N\_Vector*, *sundials4py.core.\_generic\_N\_Vector*, *typing\_extensions.CapsuleType*], *int*], *t0*: *float*, *y0*: *sundials4py.core.\_generic\_N\_Vector*, *sunctx*: *sundials4py.core.SUNContext\_*) → *sundials4py.arkode.ARKodeView*



See [`SPRKStepCreate\(\)`](#).

`sundials4py.arkode.SPRKStepGetCurrentMethod(arkode_mem: typing_extensions.CapsuleType) → tuple[int, sundials4py.arkode.ARKodeSPRKTableMem]`

`nb::rv_policy::reference`

See [`SPRKStepGetCurrentMethod\(\)`](#).

`sundials4py.arkode.SPRKStepSetMethod(arkode_mem: typing_extensions.CapsuleType, sprk_storage: sundials4py.arkode.ARKodeSPRKTableMem) → int`

See [`SPRKStepSetMethod\(\)`](#).

`sundials4py.arkode.SPRKStepSetMethodName(arkode_mem: typing_extensions.CapsuleType, method: str) → int`

See [`SPRKStepSetMethodName\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_Copy(coefficients: sundials4py.arkode.SplittingStepCoefficientsMem) → sundials4py.arkode.SplittingStepCoefficientsMem`

See [`SplittingStepCoefficients\_Copy\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_Create(sequential_methods: int, stages: int, partitions: int, order: int, alpha_1d: numpy.ndarray[dtype=float64, shape=(*)], order='C'], beta_1d: numpy.ndarray[dtype=float64, shape=(*)], order='C']) → sundials4py.arkode.SplittingStepCoefficientsMem`

See [`SplittingStepCoefficients\_Create\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_IDToName(id: sundials4py.arkode.ARKODE_SplittingCoefficientsID) → str`

See [`SplittingStepCoefficients\_IDToName\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_LieTrotter(partitions: int) → sundials4py.arkode.SplittingStepCoefficientsMem`

See [`SplittingStepCoefficients\_LieTrotter\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_LoadCoefficients(id: sundials4py.arkode.ARKODE_SplittingCoefficientsID) → sundials4py.arkode.SplittingStepCoefficientsMem`

See [`SplittingStepCoefficients\_LoadCoefficients\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_LoadCoefficientsByName(name: str) → sundials4py.arkode.SplittingStepCoefficientsMem`

See [`SplittingStepCoefficients\_LoadCoefficientsByName\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_Parallel(partitions: int) → sundials4py.arkode.SplittingStepCoefficientsMem`

See [`SplittingStepCoefficients\_Parallel\(\)`](#).

`sundials4py.arkode.SplittingStepCoefficients_Strang(partitions: int) → sundi-  
als4py.arkode.SplittingStepCoefficientsMem`

See `SplittingStepCoefficients_Strang()`.

`sundials4py.arkode.SplittingStepCoefficients_SuzukiFractal(partitions: int, order: int) → sundi-  
als4py.arkode.SplittingStepCoefficientsMem`

See `SplittingStepCoefficients_SuzukiFractal()`.

`sundials4py.arkode.SplittingStepCoefficients_SymmetricParallel(partitions: int) → sundi-  
als4py.arkode.SplittingStepCoefficientsMem`

See `SplittingStepCoefficients_SymmetricParallel()`.

`sundials4py.arkode.SplittingStepCoefficients_ThirdOrderSuzuki(partitions: int) → sundi-  
als4py.arkode.SplittingStepCoefficientsMem`

See `SplittingStepCoefficients_ThirdOrderSuzuki()`.

`sundials4py.arkode.SplittingStepCoefficients_TripleJump(partitions: int, order: int) → sundi-  
als4py.arkode.SplittingStepCoefficientsMem`

See `SplittingStepCoefficients_TripleJump()`.

`sundials4py.arkode.SplittingStepCoefficients_Write(coefficients:  
sundials4py.arkode.SplittingStepCoefficientsMem,  
outfile: sundials4py.core.FILE) → None`

See `SplittingStepCoefficients_Write()`.

`sundials4py.arkode.SplittingStepCreate(stepbers:  
collections.abc.Sequence[sundials4py.core.SUNStepper_],  
partitions: int, t0: float, y0: sundials4py.core._generic_N_Vector,  
sunctx: sundials4py.core.SUNContext_) →  
sundials4py.arkode.ARKodeView`

See `SplittingStepCreate()`.

`sundials4py.arkode.SplittingStepGetNumEvolves(arkode_mem: typing_extensions.CapsuleType, partition:  
int) → tuple[int, int]`

See `SplittingStepGetNumEvolves()`.

`sundials4py.arkode.SplittingStepReInit(arkode_mem: typing_extensions.CapsuleType, stepbers:  
collections.abc.Sequence[sundials4py.core.SUNStepper_],  
partitions: int, t0: float, y0:  
sundials4py.core._generic_N_Vector) → int`

See `SplittingStepReInit()`.

`sundials4py.arkode.SplittingStepSetCoefficients(arkode_mem: typing_extensions.CapsuleType,  
coefficients:  
sundials4py.arkode.SplittingStepCoefficientsMem) →  
int`

See `SplittingStepSetCoefficients()`.

## Chapter 22

# Release History

Date	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
Apr 2026	<a href="#">7.7.0</a>	6.7.0	7.7.0	7.7.0	7.7.0	6.7.0	7.7.0
Jan 2026	<a href="#">7.6.0</a>	6.6.0	7.6.0	7.6.0	7.6.0	6.6.0	7.6.0
Sep 2025	<a href="#">7.5.0</a>	6.5.0	7.5.0	7.5.0	7.5.0	6.5.0	7.5.0
Jun 2025	<a href="#">7.4.0</a>	6.4.0	7.4.0	7.4.0	7.4.0	6.4.0	7.4.0
Apr 2025	<a href="#">7.3.0</a>	6.3.0	7.3.0	7.3.0	7.3.0	6.3.0	7.3.0
Dec 2024	<a href="#">7.2.1</a>	6.2.1	7.2.1	7.2.1	7.2.1	6.2.1	7.2.1
Dec 2024	<a href="#">7.2.0</a>	6.2.0	7.2.0	7.2.0	7.2.0	6.2.0	7.2.0
Jun 2024	<a href="#">7.1.1</a>	6.1.1	7.1.1	7.1.1	7.1.1	6.1.1	7.1.1
Jun 2024	<a href="#">7.1.0</a>	6.1.0	7.1.0	7.1.0	7.1.0	6.1.0	7.1.0
Feb 2024	<a href="#">7.0.0</a>	6.0.0	7.0.0	7.0.0	7.0.0	6.0.0	7.0.0
Dec 2023	<a href="#">6.7.0</a>	5.7.0	6.7.0	6.7.0	6.7.0	5.7.0	6.7.0
Nov 2023	<a href="#">6.6.2</a>	5.6.2	6.6.2	6.6.2	6.6.2	5.6.2	6.6.2
Sep 2023	<a href="#">6.6.1</a>	5.6.1	6.6.1	6.6.1	6.6.1	5.6.1	6.6.1
Jul 2023	<a href="#">6.6.0</a>	5.6.0	6.6.0	6.6.0	6.6.0	5.6.0	6.6.0
Mar 2023	<a href="#">6.5.1</a>	5.5.1	6.5.1	6.5.1	6.5.1	5.5.1	6.5.1
Dec 2022	<a href="#">6.5.0</a>	5.5.0	6.5.0	6.5.0	6.5.0	5.5.0	6.5.0
Oct 2022	<a href="#">6.4.1</a>	5.4.1	6.4.1	6.4.1	6.4.1	5.4.1	6.4.1
Oct 2022	<a href="#">6.4.0</a>	5.4.0	6.4.0	6.4.0	6.4.0	5.4.0	6.4.0
Aug 2022	<a href="#">6.3.0</a>	5.3.0	6.3.0	6.3.0	6.3.0	5.3.0	6.3.0
Apr 2022	<a href="#">6.2.0</a>	5.2.0	6.2.0	6.2.0	6.2.0	5.2.0	6.2.0
Feb 2022	<a href="#">6.1.1</a>	5.1.1	6.1.1	6.1.1	6.1.1	5.1.1	6.1.1
Jan 2022	<a href="#">6.1.0</a>	5.1.0	6.1.0	6.1.0	6.1.0	5.1.0	6.1.0
Dec 2021	<a href="#">6.0.0</a>	5.0.0	6.0.0	6.0.0	6.0.0	5.0.0	6.0.0
Sep 2021	<a href="#">5.8.0</a>	4.8.0	5.8.0	5.8.0	5.8.0	4.8.0	5.8.0
Jan 2021	<a href="#">5.7.0</a>	4.7.0	5.7.0	5.7.0	5.7.0	4.7.0	5.7.0
Dec 2020	<a href="#">5.6.1</a>	4.6.1	5.6.1	5.6.1	5.6.1	4.6.1	5.6.1
Dec 2020	<a href="#">5.6.0</a>	4.6.0	5.6.0	5.6.0	5.6.0	4.6.0	5.6.0
Oct 2020	<a href="#">5.5.0</a>	4.5.0	5.5.0	5.5.0	5.5.0	4.5.0	5.5.0
Sep 2020	<a href="#">5.4.0</a>	4.4.0	5.4.0	5.4.0	5.4.0	4.4.0	5.4.0
May 2020	<a href="#">5.3.0</a>	4.3.0	5.3.0	5.3.0	5.3.0	4.3.0	5.3.0
Mar 2020	<a href="#">5.2.0</a>	4.2.0	5.2.0	5.2.0	5.2.0	4.2.0	5.2.0
Jan 2020	<a href="#">5.1.0</a>	4.1.0	5.1.0	5.1.0	5.1.0	4.1.0	5.1.0
Oct 2019	<a href="#">5.0.0</a>	4.0.0	5.0.0	5.0.0	5.0.0	4.0.0	5.0.0
Feb 2019	<a href="#">4.1.0</a>	3.1.0	4.1.0	4.1.0	4.1.0	3.1.0	4.1.0

continues on next page

Table 22.1 – continued from previous page

Date	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
Jan 2019	<a href="#">4.0.2</a>	3.0.2	4.0.2	4.0.2	4.0.2	3.0.2	4.0.2
Dec 2018	<a href="#">4.0.1</a>	3.0.1	4.0.1	4.0.1	4.0.1	3.0.1	4.0.1
Dec 2018	<a href="#">4.0.0</a>	3.0.0	4.0.0	4.0.0	4.0.0	3.0.0	4.0.0
Oct 2018	<a href="#">3.2.1</a>	2.2.1	3.2.1	3.2.1	3.2.1	2.2.1	3.2.1
Sep 2018	<a href="#">3.2.0</a>	2.2.0	3.2.0	3.2.0	3.2.0	2.2.0	3.2.0
Jul 2018	<a href="#">3.1.2</a>	2.1.2	3.1.2	3.1.2	3.1.2	2.1.2	3.1.2
May 2018	<a href="#">3.1.1</a>	2.1.1	3.1.1	3.1.1	3.1.1	2.1.1	3.1.1
Nov 2017	<a href="#">3.1.0</a>	2.1.0	3.1.0	3.1.0	3.1.0	2.1.0	3.1.0
Sep 2017	<a href="#">3.0.0</a>	2.0.0	3.0.0	3.0.0	3.0.0	2.0.0	3.0.0
Sep 2016	<a href="#">2.7.0</a>	1.1.0	2.9.0	2.9.0	2.9.0	1.3.0	2.9.0
Aug 2015	<a href="#">2.6.2</a>	1.0.2	2.8.2	2.8.2	2.8.2	1.2.2	2.8.2
Mar 2015	<a href="#">2.6.1</a>	1.0.1	2.8.1	2.8.1	2.8.1	1.2.1	2.8.1
Mar 2015	<a href="#">2.6.0</a>	1.0.0	2.8.0	2.8.0	2.8.0	1.2.0	2.8.0
Mar 2012	<a href="#">2.5.0</a>	–	2.7.0	2.7.0	2.7.0	1.1.0	2.7.0
May 2009	<a href="#">2.4.0</a>	–	2.6.0	2.6.0	2.6.0	1.0.0	2.6.0
Nov 2006	<a href="#">2.3.0</a>	–	2.5.0	2.5.0	2.5.0	–	2.5.0
Mar 2006	<a href="#">2.2.0</a>	–	2.4.0	2.4.0	2.4.0	–	2.4.0
May 2005	<a href="#">2.1.1</a>	–	2.3.0	2.3.0	2.3.0	–	2.3.0
Apr 2005	<a href="#">2.1.0</a>	–	2.3.0	2.2.0	2.3.0	–	2.3.0
Mar 2005	<a href="#">2.0.2</a>	–	2.2.2	2.1.2	2.2.2	–	2.2.2
Jan 2005	<a href="#">2.0.1</a>	–	2.2.1	2.1.1	2.2.1	–	2.2.1
Dec 2004	<a href="#">2.0.0</a>	–	2.2.0	2.1.0	2.2.0	–	2.2.0
Jul 2002	1.0.0	–	2.0.0	1.0.0	2.0.0	–	2.0.0
Mar 2002	–	–	1.0.0 <sup>3</sup>	–	–	–	–
Feb 1999	–	–	–	–	1.0.0 <sup>4</sup>	–	–
Aug 1998	–	–	–	–	–	–	1.0.0 <sup>5</sup>
Jul 1997	–	–	1.0.0 <sup>2</sup>	–	–	–	–
Sep 1994	–	–	1.0.0 <sup>1</sup>	–	–	–	–

1. CVODE written
2. PVODE written
3. CVODE and PVODE combined
4. IDA written
5. KINSOL written

# Chapter 23

## Changelog

### 23.1 Changes to SUNDIALS in release 7.7.0

#### New Features and Enhancements

The default number of stages for the SSP Runge-Kutta methods `ARKODE_LSRK_SSP_S_2` and `ARKODE_LSRK_SSP_S_3` in `LSRKStep` were changed from 10 and 9, respectively, to their minimum allowable values of 2 and 4. Users may revert to the previous values by calling `LSRKStepSetNumSSPStages()`.

Added the optional function `ARKodeInit()` to ARKODE to enable data allocation before the first call to `ARKodeEvolve()` (but after all other optional input routines have been called), to support users who measure memory usage before beginning a simulation.

Added the function `ARKodeGetStageIndex()` that returns the index of the stage currently being processed, and the total number of stages in the method, for users who wish to compute auxiliary quantities in their IVP right-hand side functions during some stages and not others (e.g., in all but the first or last stage).

Added the functions `ARKodeGetLastTime()` and `ARKodeGetLastState()` to return the last successful time and state achieved by ARKODE, respectively.

ARKODE now allows users to supply functions that will be called before each internal time step attempt (`ARKodeSetPreStepFn()`), after each successful time step (`ARKodeSetPostStepFn()`), before right-hand side routines are called on an updated state (`ARKodeSetPreRhsFn()`), and/or once each internal step/stage is computed (`ARKodeSetPostprocessStepFn()`/`ARKodeSetPostprocessStageFn()`). These are considered **advanced** functions, as they should treat the state vector as read-only, otherwise all theoretical guarantees of solution accuracy and stability will be lost. As a result of these new functions, the values of multiple ARKODE return codes (e.g., `ARK_INTERP_FAIL`) have been updated; users who key off of the named constants will not be affected, but users who rely on the values themselves should update their codes accordingly.

Note to users utilizing the previously undocumented `ARKodeSetPostprocessStepFn()` function, the supplied function is now called on the newly computed state vector for all step attempts not just successful steps. To obtain the previous behavior of only calling a function on successful steps, switch to using `ARKodeSetPostStepFn()`.

Added `SUNLogger_Set{Error,Warning,Info,Debug}File` functions to allow setting logger output streams with a `FILE*`.

Updated the Kokkos `N_Vector` to support Kokkos 5.x versions.

#### Bug Fixes

Fixed a CMake bug where the SuperLU\_MT interface would not be built and installed without setting the `SUPERLUMT_WORKS` option to `TRUE`.

Fixed the embedded coefficients for the ARKODE\_TSITOURAS\_7\_4\_5 Butcher table.

Fixed a bug in LSRKStep where an incorrect state vector could be passed to a user-supplied dominant eigenvalue function on the first step unless the output vector passed to [ARKodeEvolve\(\)](#) contained the initial condition and when an eigenvalue estimate is requested on the first step in a subsequent call to [ARKodeEvolve\(\)](#) unless the output vector passed contained the most recently returned solution.

Fixed a potential bug in LSRKStep's [ARKODE\\_LSRK\\_SSP\\_S\\_3](#) method, where a real number was used instead of an integer, potentially resulting in a rounding error.

Fixed a bug in MRIStep for estimating the first “slow” time step in an adaptive multirate calculation.

Fixed a bug in MRIStep when using a custom inner integrator that relies on the input state being the initial condition for the fast integration rather than retaining the result from the last inner integration or most recent reset call and the output vector passed to [ARKodeEvolve\(\)](#) does not contain the initial condition on the first call or the last returned solution on subsequent calls.

Added a missing call to [SUNNonlinSolSetup\(\)](#) in MRIStep when using an IMEX-MRI-SR method.

Fixed a bug in the ARKODE discrete adjoint checkpointing where an incorrect state would be stored on the first step if the output vector passed to [ARKodeEvolve\(\)](#) did not contain the initial condition on the first call.

Removed extraneous copy of output vector when using ARKODE in ARK\_ONE\_STEP mode.

Removed an extraneous copy of the output vector in each step with SplittingStep.

Fixed a bug in logging output from ARKODE, where for some time stepping modules, the current “time” output in the logger was incorrect.

Fixed a bug where passing an empty string to `SUNLogger_Set{Error,Warning,Info,Debug}Filename` did not disable the corresponding logging stream [Issue #844](#).

## Deprecation Notices

The `CCodeSetMonitorFn` and `CCodeSetMonitorFrequency` functions have been deprecated and will be removed in the next major release.

Several CMake options have been deprecated in favor of namespaced versions prefixed with `SUNDIALS_` to avoid naming collisions in applications that include SUNDIALS directly within their CMake builds. Additionally, a consistent naming convention (`SUNDIALS_ENABLE`) is now used for all boolean options. The table below lists the old CMake option names and the new replacements.

Old Option	New Option
BUILD_ARKODE	<a href="#">SUNDIALS_ENABLE_ARKODE</a>
BUILD_CCODE	<a href="#">SUNDIALS_ENABLE_CCODE</a>
BUILD_CCODES	<a href="#">SUNDIALS_ENABLE_CCODES</a>
BUILD_IDA	<a href="#">SUNDIALS_ENABLE_IDA</a>
BUILD_IDAS	<a href="#">SUNDIALS_ENABLE_IDAS</a>
BUILD_KINSOL	<a href="#">SUNDIALS_ENABLE_KINSOL</a>
ENABLE_MPI	<a href="#">SUNDIALS_ENABLE_MPI</a>
ENABLE_OPENMP	<a href="#">SUNDIALS_ENABLE_OPENMP</a>
ENABLE_OPENMP_DEVICE	<a href="#">SUNDIALS_ENABLE_OPENMP_DEVICE</a>
OPENMP_DEVICE_WORKS	<a href="#">SUNDIALS_ENABLE_OPENMP_DEVICE_CHECKS</a>
ENABLE_PTHREAD	<a href="#">SUNDIALS_ENABLE_PTHREAD</a>
ENABLE_CUDA	<a href="#">SUNDIALS_ENABLE_CUDA</a>
ENABLE_HIP	<a href="#">SUNDIALS_ENABLE_HIP</a>
ENABLE_SYCL	<a href="#">SUNDIALS_ENABLE_SYCL</a>
ENABLE_LAPACK	<a href="#">SUNDIALS_ENABLE_LAPACK</a>
LAPACK_WORKS	<a href="#">SUNDIALS_ENABLE_LAPACK_CHECKS</a>

continues on next page

Table 23.1 – continued from previous page

ENABLE_GINKGO	<i>SUNDIALS_ENABLE_GINKGO</i>
GINKGO_WORKS	<i>SUNDIALS_ENABLE_GINKGO_CHECKS</i>
ENABLE_MAGMA	<i>SUNDIALS_ENABLE_MAGMA</i>
MAGMA_WORKS	<i>SUNDIALS_ENABLE_MAGMA_CHECKS</i>
ENABLE_SUPERLUDIST	<i>SUNDIALS_ENABLE_SUPERLUDIST</i>
SUPERLUDIST_WORKS	<i>SUNDIALS_ENABLE_SUPERLUDIST_CHECKS</i>
ENABLE_SUPERLUMT	<i>SUNDIALS_ENABLE_SUPERLUMT</i>
SUPERLUMT_WORKS	<i>SUNDIALS_ENABLE_SUPERLUMT_CHECKS</i>
ENABLE_KLU	<i>SUNDIALS_ENABLE_KLU</i>
KLU_WORKS	<i>SUNDIALS_ENABLE_KLU_CHECKS</i>
ENABLE_HYPRE	<i>SUNDIALS_ENABLE_HYPRE</i>
HYPRE_WORKS	<i>SUNDIALS_ENABLE_HYPRE_CHECKS</i>
ENABLE_PETSC	<i>SUNDIALS_ENABLE_PETSC</i>
PETSC_WORKS	<i>SUNDIALS_ENABLE_PETSC_CHECKS</i>
ENABLE_TRILINOS	<i>SUNDIALS_ENABLE_TRILINOS</i>
ENABLE_RAJA	<i>SUNDIALS_ENABLE_RAJA</i>
ENABLE_XBRAID	<i>SUNDIALS_ENABLE_XBRAID</i>
XBRAID_WORKS	<i>SUNDIALS_ENABLE_XBRAID_CHECKS</i>
ENABLE_ONEMKL	<i>SUNDIALS_ENABLE_ONEMKL</i>
ONEMKL_WORKS	<i>SUNDIALS_ENABLE_ONEMKL_CHECKS</i>
ENABLE_CALIPER	<i>SUNDIALS_ENABLE_CALIPER</i>
ENABLE_ADIK	<i>SUNDIALS_ENABLE_ADIK</i>
ENABLE_KOKKOS	<i>SUNDIALS_ENABLE_KOKKOS</i>
KOKKOS_WORKS	<i>SUNDIALS_ENABLE_KOKKOS_CHECKS</i>
ENABLE_KOKKOS_KERNELS	<i>SUNDIALS_ENABLE_KOKKOS_KERNELS</i>
KOKKOS_KERNELS_WORKS	<i>SUNDIALS_ENABLE_KOKKOS_KERNELS_CHECKS</i>
BUILD_FORTRAN_MODULE_INTERFACE	<i>SUNDIALS_ENABLE_FORTRAN</i>
SUNDIALS_BUILD_WITH_PROFILING	<i>SUNDIALS_ENABLE_PROFILING</i>
SUNDIALS_BUILD_WITH_MONITORING	<i>SUNDIALS_ENABLE_MONITORING</i>
SUNDIALS_BUILD_PACKAGE_FUSED_KERNELS	<i>SUNDIALS_ENABLE_PACKAGE_FUSED_KERNELS</i>
EXAMPLES_ENABLE_C	<i>SUNDIALS_ENABLE_C_EXAMPLES</i>
EXAMPLES_ENABLE_CXX	<i>SUNDIALS_ENABLE_CXX_EXAMPLES</i>
EXAMPLES_ENABLE_F2003	<i>SUNDIALS_ENABLE_FORTRAN_EXAMPLES</i>
EXAMPLES_ENABLE_CUDA	<i>SUNDIALS_ENABLE_CUDA_EXAMPLES</i>
EXAMPLES_INSTALL	<i>SUNDIALS_ENABLE_EXAMPLES_INSTALL</i>
EXAMPLES_INSTALL_PATH	<i>SUNDIALS_EXAMPLES_INSTALL_PATH</i>
BUILD_BENCHMARKS	<i>SUNDIALS_ENABLE_BENCHMARKS</i>
BENCHMARKS_INSTALL_PATH	<i>SUNDIALS_BENCHMARKS_INSTALL_PATH</i>
SUNDIALS_BENCHMARK_OUTPUT_DIR	<i>SUNDIALS_BENCHMARKS_OUTPUT_DIR</i>
SUNDIALS_BENCHMARK_CALIPER_OUTPUT_DIR	<i>SUNDIALS_BENCHMARKS_CALIPER_OUTPUT_DIR</i>
SUNDIALS_BENCHMARK_NUM_CPUS	<i>SUNDIALS_BENCHMARKS_NUM_CPUS</i>
SUNDIALS_BENCHMARK_NUM_GPUS	<i>SUNDIALS_BENCHMARKS_NUM_GPUS</i>
ENABLE_ALL_WARNINGS	<i>SUNDIALS_ENABLE_ALL_WARNINGS</i>
ENABLE_WARNINGS_AS_ERRORS	<i>CMAKE_COMPILE_WARNING_AS_ERROR</i>
ENABLE_ADDRESS_SANITIZER	<i>SUNDIALS_ENABLE_ADDRESS_SANITIZER</i>
ENABLE_MEMORY_SANITIZER	<i>SUNDIALS_ENABLE_MEMORY_SANITIZER</i>
ENABLE_LEAK_SANITIZER	<i>SUNDIALS_ENABLE_LEAK_SANITIZER</i>

Following the updated CMake options, the macros listed below have been deprecated and replaced with versions that align with the new CMake options.



Old Macro	New Macro
SUNDIALS_BUILD_WITH_PROFILING	SUNDIALS_ENABLE_PROFILING
SUNDIALS_BUILD_WITH_MONITORING	SUNDIALS_ENABLE_MONITORING
SUNDIALS_BUILD_PACKAGE_FUSED_KERNELS	SUNDIALS_ENABLE_PACKAGE_FUSED_KERNELS

## 23.2 Changes to SUNDIALS in release 7.6.0

### Major Features

SUNDIALS now has official Python interfaces! With this release, we are shipping a **beta version** of the `sundials4py` Python module (created with `nanobind` and `litgen`). `sundials4py` provides explicit interfaces to most features of SUNDIALS. See the [Python](#) section of the user guide for more information.

### New Features and Enhancements

Added functions to `CVODE(S)` and `IDA(S)` to set the maximum number of inequality constraint failures in a step attempt (`CVodeSetMaxNumConstraintFails()` and `IDASetMaxNumConstraintFails()`) and to retrieve the total number of failed step attempts due to an inequality constraint violation (`CVodeGetNumConstraintFails()` and `IDAGetNumConstraintFails()`). As a result, constraint failures are no longer included in the number of step failures due to a solver failure (i.e., the values returned by `CVodeGetNumStepSolveFails()` and `IDAGetNumStepSolveFails()`). The functions `CVodeGetNumConstraintCorrections()` and `IDAGetNumConstraintCorrections()` were also added to retrieve the number of steps where the corrector was modified to satisfy an inequality constraint without failing the step.

The functions `CVodeGetUserDataB` and `IDAGetUserDataB` were added to `CVODES` and `IDAS`, respectively.

### Bug Fixes

Fixed a bug in the `CVODE(S)` inequality constraint handling where the predicted state was used to compute the step size reduction factor which could lead to an insufficient reduction in the step size or, when the prediction violates the constraints, an infinitely large step size in the next step attempt ([Issue #702](#)).

On the initial time step with a user-supplied initial step size, `ARKODE` and `CVODE(S)` will now return `ARK_T00_CLOSE` or `CV_T00_CLOSE`, respectively, when the requested output time is the same as, or within numerical roundoff of, the initial time ([Issue #722](#)). Before a `T00_CLOSE` error would only be returned when internally estimating the initial step size. In `IDA(S)`, added a `IDA_T00_CLOSE` return value for when the initial and output time are too close. Previously, `IDA(S)` would return `IDA_ILL_INPUT`.

Fixed a bug in `ARKODE`, `CVODE(S)`, and `IDA(S)` where the linear solver counters were not reset on reinitialization until the next call to advance the system. As such, non-zero linear solver statistics could be returned if retrieving or printing linear solver counters between reinitialization and the next call to advance the system.

In `CVODES` and `IDA`, added missing return flag names to `CVodeGetReturnFlagName()` and `IDAGetReturnFlagName()`, respectively.

The `SPRKStep` module now accounts for zero coefficients in the `SPRK` tables, eliminating extraneous function evaluations.

A bug was fixed in `KINSOL` where the information logging function would always be called even when informational logging was disabled ([Issue #801](#)).

A bug preventing a user supplied `SUNStepper_ResetCheckpointIndex()` function from being called was fixed.

The interface to Ginkgo batched linear solvers has been updated to fix build errors when using 64-bit index types ([Issue #797](#)). Note, only the batched dense matrix in Ginkgo is currently compatible with 64-bit indexing (as of Ginkgo 1.10).

The Kokkos `N_Vector` now properly handles unmanaged views. Previously, if a Kokkos `N_Vector` was created from an unmanaged view, the view would become a managed view and the data would be freed unexpectedly.



Fixed a CMake bug which resulted in static targets depending on shared targets when building both types of libraries in the same build ([Issue #692](#)).

Some installed Fortran example makefiles were not linking to `sundials_fcore_mod` and `sundials_core` libraries as they should be. This is now fixed.

### Deprecation Notices

The `N_Vector_S` typedef to `N_Vector*` is deprecated and will be removed in the next major release.

The `CSC_MAT` and `CSR_MAT` macros defined in `sunmatrix_sparse.h` will be removed in the next major release. Use `SUN_CSC_MAT` and `SUN_CSR_MAT` instead.

`SUNDIALSFileOpen` and `SUNDIALSFileClose` will be removed in the next major release. Use [`SUNFileOpen\(\)`](#) and [`SUNFileClose\(\)`](#) instead.

The `Convert` methods on the `sundials::kokkos::Vector`, `sundials::kokkos::DenseMatrix`, `sundials::ginkgo::Matrix`, `sundials::ginkgo::BatchMatrix`, `sundials::kokkos::DenseLinearSolver`, `sundials::ginkgo::LinearSolver`, and `sundials::ginkgo::BatchLinearSolver` classes have been deprecated and will be removed in the next major release. The method `get`, should be used instead.

## 23.3 Changes to SUNDIALS in release 7.5.0

### Major Features

Added the [`SUNDomEigEstimator`](#) interface for estimating the dominant eigenvalue value of a system. Two implementations are provided: Power Iteration and Arnoldi Iteration. The latter method requires building with LAPACK support enabled.

Added the function [`LSRKStepSetDomEigEstimator`](#) in `LSRKStep` to attach a [`SUNDomEigEstimator`](#), when using Runge-Kutta-Chebyshev or Runge-Kutta-Legendre methods, as an alternative to supplying a user-defined function to compute the dominant eigenvalue.

Added `SetOptions` functions all SUNDIALS packages and the classes for adaptivity controllers, dominant eigenvalue estimators, linear solvers, and nonlinear solvers to support setting options with command line inputs.

### New Features and Enhancements

A new `SUNLinearSolver`, `SUNLINEARSOLVER_GINKGOBATCH`, and corresponding `SUNMatrix`, `SUNMATRIX_GINKGOBATCH`, were added for solving block/batched linear systems with the [Ginkgo linear solver library](#). As a result, Ginkgo 1.9.0 or newer is now required when enabling Ginkgo support.

The functions [`KINSetMAA\(\)`](#) and [`KINSetOrthAA\(\)`](#) have been updated to allow for setting the Anderson acceleration depth and orthogonalization method after [`KINInit\(\)`](#). Additionally, [`KINSetMAA\(\)`](#) and [`KINSetNumMaxIters\(\)`](#) may now be called in any order.

### Bug Fixes

Fixed a bug in how `MRIStep` interacts with an `MRIHTol` `SUNAdaptController` object (the previous version essentially just reverted to a decoupled multirate controller). Removed the upper limit on `inner_max_tolfac` in [`SUNAdaptController\_SetParams\_MRIHTol\(\)`](#).

The shared library version numbers for the oneMKL dense linear solver and matrix as well as the PETSc SNES nonlinear solver have been corrected.

Fixed a CMake bug where the MRI H-Tol controller was not included in the ARKODE Fortran module.

Fixed a bug in the CUDA and HIP implementations of [`SUNMemoryHelper\_CopyAsync\(\)`](#) where the execution stream is not extracted correctly from the helper when a stream is not provided to [`SUNMemoryHelper\_CopyAsync\(\)`](#).

Fixed a bug in MRISStep where a segfault would occur when an MRI coupling table is not explicitly set and an MRI integrator is nested inside another MRI integrator.

Fixed a bug in MRISStep where MERK methods with unordered stage groups (MERK43 and MERK54) would include stage right-hand side vectors that had not been computed yet in fast time scale forcing computations. These vectors were scaled by zero, so in most cases the extraneous computations would not impact results. However, in cases where these vectors contain `inf` or `nan`, this would lead to erroneous forcing terms.

Fixed a bug in [`ARKodeSetDefaults\(\)`](#) with LSRKStep where the stored spectral radius data was reset to zero, flags to update the dominant eigenvalue were reset to true, and a flag indicating if an SSP is being used was reset to false.

Fixed a bug introduced in v7.3.0 in KINSOL when using Anderson acceleration and solving a problem multiple times with the same KINSOL instance. In this use case, the current Anderson acceleration depth from the initial solve was not reinitialized on subsequent solves.

Fixed a logging bug in KINSOL where logging messages would not be output.

Fixed a bug in the `suntools.logs` Python module where the `get_history` function, when given a `step_status` for filtering output from a multirate method, would only extract values from the fast time scale if the slow time scale step matched the given status filter. Fixed an additional bug in `get_history` with MRI-GARK methods where values would not be extracted from a fast time scale integration associated with an embedding.

## 23.4 Changes to SUNDIALS in release 7.4.0

### New Features and Enhancements

[`ARKodeSetCFLFraction\(\)`](#) now allows `cfl_frac` to be greater than or equal to one.

Added an option to enable compensated summation of the time accumulator for all of ARKODE. This was previously only an option for the SPRKStep module. The new function to call to enable this is [`ARKodeSetUseCompensatedSums\(\)`](#).

### Bug Fixes

Fixed segfaults in [`CVodeAdjInit\(\)`](#) and [`IDAAdjInit\(\)`](#) when called after adjoint memory has been freed.

Fixed a CMake bug that would cause the Caliper compile test to fail at configure time.

Fixed a bug in the CVODE/CVODES [`CVodeSetEtaFixedStepBounds\(\)`](#) function which disallowed setting `eta_min_fx` or `eta_max_fx` to 1.

[`SUNAdjointStepper\_PrintAllStats\(\)`](#) was reporting the wrong quantity for the number of “recompute passes” and has been fixed.

### Deprecation Notices

The [`SPRKStepSetUseCompensatedSums\(\)`](#) function has been deprecated. Use the [`ARKodeSetUseCompensatedSums\(\)`](#) function instead.

## 23.5 Changes to SUNDIALS in release 7.3.0

### Major Features

A new discrete adjoint capability for explicit Runge–Kutta methods has been added to the ARKODE ERKStep and ARKStep stepper modules. This is based on a new set of shared classes, [`SUNAdjointStepper`](#) and [`SUNAdjointCheckpointScheme`](#). A new example demonstrating this capability can be found in `examples/arkode/C_serial/ark_lotka_volterra_ASA.c`. See the [Adjoint Sensitivity Analysis](#) section of the ARKODE user guide for details.

## New Features and Enhancements

### ARKODE

The following changes have been made to the default ERK, DIRK, and ARK methods in ARKODE to utilize more efficient methods:

Type	Old Default	New Default
2nd Order Explicit	ARKODE_HEUN_EULER_2_1_2	ARKODE_RALSTON_3_1_2
4th Order Explicit	ARKODE_ZONNEVELD_5_3_4	ARKODE_SOFRONIOU_SPALETTA_5_3_4
5th Order Explicit	ARKODE_CASH_KARP_6_4_5	ARKODE_TSITOURAS_7_4_5
6th Order Explicit	ARKODE_VERNER_8_5_6	ARKODE_VERNER_9_5_6
8th Order Explicit	ARKODE_FEHLBERG_13_7_8	ARKODE_VERNER_13_7_8
2nd Order Implicit	ARKODE_SDIRK_2_1_2	ARKODE_ARK2_DIRK_3_1_2
3rd Order Implicit	ARKODE_ARK324L2SA_DIRK_4_2_3	ARKODE_ESDIRK325L2SA_5_2_3
4th Order Implicit	ARKODE_SDIRK_5_3_4	ARKODE_ESDIRK436L2SA_6_3_4
5th Order Implicit	ARKODE_ARK548L2SA_DIRK_8_4_5	ARKODE_ESDIRK547L2SA2_7_4_5
4th Order ARK	ARKODE_ARK436L2SA_ERK_6_3_4 ARKODE_ARK436L2SA_DIRK_6_3_4	and ARKODE_ARK437L2SA_ERK_7_3_4 and ARKODE_ARK437L2SA_DIRK_7_3_4
5th Order ARK	ARKODE_ARK548L2SA_ERK_8_4_5 ARKODE_ARK548L2SA_DIRK_8_4_5	and ARKODE_ARK548L2SAb_ERK_8_4_5 and ARKODE_ARK548L2SAb_DIRK_8_4_5

The old default methods can be loaded using the functions *ERKStepSetTableName()* or *ERKStepSetTableNum()* with ERKStep and *ARKStepSetTableName()* or *ARKStepSetTableNum()* with ARKStep and passing the desired method name string or constant, respectively. For example, the following call can be used to load the old default fourth order method with ERKStep:

```
/* Load the old 4th order ERK method using the table name */
ierr = ERKStepSetTableName(arkode_mem, "ARKODE_ZONNEVELD_5_3_4");
```

Similarly with ARKStep, the following calls can be used for ERK, DIRK, or ARK methods, respectively:

```
/* Load the old 4th order ERK method by name */
ierr = ARKStepSetTableName(arkode_mem, "ARKODE_DIRK_NONE",
                           "ARKODE_ZONNEVELD_5_3_4");

/* Load the old 4th order DIRK method by name */
ierr = ARKStepSetTableName(arkode_mem, "ARKODE_SDIRK_5_3_4",
                           "ARKODE_ERK_NONE");

/* Load the old 4th order ARK method by name */
ierr = ARKStepSetTableName(arkode_mem, "ARKODE_ARK436L2SA_DIRK_6_3_4",
                           "ARKODE_ARK436L2SA_ERK_6_3_4");
```

Additionally, the following changes have been made to the default time step adaptivity parameters in ARKODE:

Parameter	Old Default	New Default
Controller	PID (PI for ERKStep)	I
Safety Factor	0.96	0.9
Bias	1.5 (1.2 for ERKStep)	1.0
Fixed Step Bounds	[1.0, 1.5]	[1.0, 1.0]
Adaptivity Adjustment	-1	0

The following calls can be used to restore the old defaults for ERKStep:

```
SUNAdaptController controller = SUNAdaptController_Soderlind(ctx);
SUNAdaptController_SetParams_PI(controller, 0.8, -0.31);
ARKodeSetAdaptController(arkode_mem, controller);
SUNAdaptController_SetErrorBias(controller, 1.2);
ARKodeSetSafetyFactor(arkode_mem, 0.96);
ARKodeSetFixedStepBounds(arkode_mem, 1, 1.5);
ARKodeSetAdaptivityAdjustment(arkode_mem, -1);
```

The following calls can be used to restore the old defaults for other ARKODE integrators:

```
SUNAdaptController controller = SUNAdaptController_PID(ctx);
ARKodeSetAdaptController(arkode_mem, controller);
SUNAdaptController_SetErrorBias(controller, 1.5);
ARKodeSetSafetyFactor(arkode_mem, 0.96);
ARKodeSetFixedStepBounds(arkode_mem, 1, 1.5);
ARKodeSetAdaptivityAdjustment(arkode_mem, -1);
```

In both cases above, destroy the controller at the end of the run with `SUNAdaptController_Destroy(controller)`;

The Soderlind time step adaptivity controller now starts with an I controller until there is sufficient history of past time steps and errors.

Added `ARKodeSetAdaptControllerByName()` to set a time step adaptivity controller with a string. There are also four new controllers: `SUNAdaptController_H0211()`, `SUNAdaptController_H0321()`, `SUNAdaptController_H211()`, and `SUNAdaptController_H312()`.

Added the ARKODE\_RALSTON\_3\_1\_2 and ARKODE\_TSITOURAS\_7\_4\_5 explicit Runge-Kutta Butcher tables.

Improved the precision of the coefficients for ARKODE\_ARK324L2SA\_ERK\_4\_2\_3, ARKODE\_VERNER\_9\_5\_6, ARKODE\_VERNER\_10\_6\_7, ARKODE\_VERNER\_13\_7\_8, ARKODE\_ARK324L2SA\_DIRK\_4\_2\_3, and ARKODE\_ESDIRK324L2SA\_4\_2\_3.

### CVODE / CVODES

Added support for resizing CVODE and CVODES when solving initial value problems where the number of equations and unknowns changes over time. Resizing requires a user supplied history of solution and right-hand side values at the new problem size, see `CVodeResizeHistory()` for more information.

### KINSOL

Added support in KINSOL for setting user-supplied functions to compute the damping factor and, when using Anderson acceleration, the depth in fixed-point or Picard iterations. See `KINSetDampingFn()` and `KINSetDepthFn()`, respectively, for more information.

### SUNDIALS Types

A new type, *suncountertype*, was added for the integer type used for counter variables. It is currently an alias for `long int`.

### Bug Fixes

#### ARKODE

Fixed bug in *ARKodeResize()* which caused it return an error for MRI methods.

Removed error floors from the *SUNAdaptController* implementations which could unnecessarily limit the time size growth, particularly after the first step.

Fixed bug in *ARKodeSetFixedStep()* where it could return `ARK_SUCCESS` despite an error occurring.

Fixed bug in the ARKODE SPRKStep *SPRKStepReInit()* function and *ARKodeReset()* function with SPRKStep that could cause a segmentation fault when compensated summation is not used.

#### KINSOL

Fixed a bug in KINSOL where an incorrect damping parameter is applied on the initial iteration with Anderson acceleration unless *KINSetDamping()* and *KINSetDampingAA()* are both called with the same value when enabling damping.

Fixed a bug in KINSOL where errors that occurred when computing Anderson acceleration were not captured.

Added missing return values to *KINGetReturnFlagName()*.

#### CMake

Fixed the behavior of *SUNDIALS\_ENABLE\_ERROR\_CHECKS* so additional runtime error checks are disabled by default with all release build types. Previously, `MinSizeRel` builds enabled additional error checking by default.

### Deprecation Notices

All work space functions, e.g., *CVodeGetWorkSpace* and *ARKodeGetLinWorkSpace*, have been deprecated and will be removed in version 8.0.0.

## 23.6 Changes to SUNDIALS in release 7.2.1

### New Features and Enhancements

Unit tests were separated from examples. To that end, the following directories were moved out of the `examples/` directory to the `test/unit_tests` directory: `nvector`, `sunmatrix`, `sunlinsol`, and `sunnonlinsol`.

### Bug Fixes

Fixed a bug in ARKStep where an extra right-hand side evaluation would occur each time step when enabling the *ARKodeSetAutonomous()* option and using an IMEX method where the DIRK table has an implicit first stage and is not stiffly accurate.

## 23.7 Changes to SUNDIALS in release 7.2.0

### Major Features

Added a time-stepping module to ARKODE for low storage Runge–Kutta methods, *LSRKStep*. This currently supports five explicit low-storage methods: the second-order Runge–Kutta–Chebyshev and Runge–Kutta–Legendre methods, and the second- through fourth-order optimal strong stability preserving Runge–Kutta methods. All methods include embeddings for temporal adaptivity.

Added an operator splitting module, *SplittingStep*, and forcing method module, *ForcingStep*, to ARKODE. These modules support a broad range of operator-split time integration methods for multiphysics applications.

Added support for multirate time step adaptivity controllers, based on the recently introduced *SUNAdaptController* base class, to ARKODE's MRIStep module. As a part of this, we added embeddings for existing MRI-GARK methods, as well as support for embedded MERK and IMEX-MRI-SR methods. Added new default MRI methods for temporally adaptive versus fixed-step runs.

## New Features and Enhancements

### *Logging*

The information level logging output in ARKODE, CVODE(S), and IDA(S) has been updated to be more uniform across the packages and a new `tools` directory has been added with a Python module, `suntools`, containing utilities for parsing logging output. The Python utilities for parsing CSV output have been relocated from the `scripts` directory to the Python module.

### *SUNStepper*

Added the *SUNStepper* base class to represent a generic solution procedure for IVPs. This is used by the *SplittingStep* and *ForcingStep* modules of ARKODE. A *SUNStepper* can be created from an ARKODE memory block with the new function *ARKodeCreateSUNStepper()*. To enable interoperability with *MRISetInnerStepper*, the function *MRISetInnerStepper\_CreateFromSUNStepper()* was added.

### *ARKODE*

Added functionality to ARKODE to accumulate a temporal error estimate over multiple time steps. See the routines *ARKodeSetAccumulatedErrorType()*, *ARKodeResetAccumulatedError()*, and *ARKodeGetAccumulatedError()* for details.

Added the *ARKodeSetStepDirection()* and *ARKodeGetStepDirection()* functions to change and query the direction of integration.

Added the function *MRISetGetNumInnerStepperFails()* to retrieve the number of recoverable failures reported by the *MRISetInnerStepper*.

Added a utility routine to wrap any valid ARKODE integrator for use as an MRIStep inner stepper object, *ARKodeCreateMRISetInnerStepper()*.

The following DIRK schemes now have coefficients accurate to quad precision:

- ARKODE\_BILLINGTON\_3\_3\_2
- ARKODE\_KVAERNO\_4\_2\_3
- ARKODE\_CASH\_5\_2\_4
- ARKODE\_CASH\_5\_3\_4
- ARKODE\_KVAERNO\_5\_3\_4
- ARKODE\_KVAERNO\_7\_4\_5

### *CMake*

The default value of *CMAKE\_CUDA\_ARCHITECTURES* is no longer set to 70 and is now determined automatically by CMake. The previous default was only valid for Volta GPUs while the automatically selected value will vary across compilers and compiler versions. As such, users are encouraged to override this value with the architecture for their system.

The build system has been updated to utilize the CMake LAPACK imported target which should ease building SUN-DIALS with LAPACK libraries that require setting specific linker flags e.g., MKL.

### *Third Party Libraries*

The Trilinos Tpetra NVector interface has been updated to utilize CMake imported targets added in Trilinos 14 to improve support for different Kokkos backends with Trilinos. As such, Trilinos 14 or newer is required and the `Trilinos_INTERFACE_*` CMake options have been removed.

Example programs using *hypre* have been updated to support v2.20 and newer.

## Bug Fixes

### *CMake*

Fixed a CMake bug regarding usage of missing “`print_warning`” macro that was only triggered when the deprecated `CUDA_ARCH` option was used.

Fixed a CMake configuration issue related to aliasing an `ALIAS` target when using `ENABLE_KLU=ON` in combination with a static-only build of SuiteSparse.

Fixed a CMake issue which caused third-party CMake variables to be unset. Users may see more options in the CMake GUI now as a result of the fix. See details in GitHub Issue #538.

### *NVector*

Fixed a build failure with the SYCL NVector when using Intel oneAPI 2025.0 compilers. See GitHub Issue #596.

Fixed compilation errors when building the Trilinos Tpetra NVector with CUDA support.

### *SUNMatrix*

Fixed a [bug](#) in the sparse matrix implementation of `SUNMatScaleAddI()` which caused out of bounds writes unless `indexvals` were in ascending order for each row/column.

### *SUNLinearSolver*

Fixed a bug in the SPTFQMR linear solver where recoverable preconditioner errors were reported as unrecoverable.

### *ARKODE*

Fixed `ARKodeResize()` not using the default `hscale` when an argument of 0 was provided.

Fixed a memory leak that could occur if `ARKodeSetDefaults()` is called repeatedly.

Fixed the loading of ARKStep’s default first order explicit method.

Fixed loading the default IMEX-MRI method if `ARKodeSetOrder()` is used to specify a third or fourth order method. Previously, the default second order method was loaded in both cases.

Fixed potential memory leaks and out of bounds array accesses that could occur in the ARKODE Lagrange interpolation module when changing the method order or polynomial degree after re-initializing an integrator.

Fixed a bug in ARKODE when enabling rootfinding with fixed step sizes and the initial value of the rootfinding function is zero. In this case, uninitialized right-hand side data was used to compute a state value near the initial condition to determine if any rootfinding functions are initially active.

Fixed a bug in MRISStep where the data supplied to the Hermite interpolation module did not include contributions from the fast right-hand side function. With this fix, users will see one additional fast right-hand side function evaluation per slow step with the Hermite interpolation option.

Fixed a bug in SPRKStep when using compensated summations where the error vector was not initialized to zero.

### *CVODE(S)*

Fixed a bug where `CVodeSetProjFailEta()` would ignore the *eta* parameter.

### *Fortran Interfaces*

Fixed a bug in the 32-bit `sunindextype` Fortran interfaces to `N_VGetSubvectorArrayPointer_ManyVector()`, `N_VGetSubvectorArrayPointer_MPIManyVector()`, `SUNBandMatrix_Column()` and `SUNDenseMatrix_Column()` where 64-bit `sunindextype` interface functions were used.



### Deprecation Notices

Deprecated the ARKStep-specific utility routine for wrapping an ARKStep instance as an MRISStep inner stepper object, [ARKStepCreateMRISStepInnerStepper\(\)](#). Use [ARKodeCreateMRISStepInnerStepper\(\)](#) instead.

The ARKODE stepper specific functions to retrieve the number of right-hand side function evaluations have been deprecated. Use [ARKodeGetNumRhsEvals\(\)](#) instead.

## 23.8 Changes to SUNDIALS in release 7.1.1

### Bug Fixes

Fixed a [bug](#) in v7.1.0 with the SYCL N\_Vector N\_VSpace function.

## 23.9 Changes to SUNDIALS in release 7.1.0

### Major Features

Created shared user interface functions for ARKODE to allow more uniform control over time-stepping algorithms, improved extensibility, and simplified code maintenance. The corresponding stepper-specific user-callable functions are now deprecated and will be removed in a future major release.

Added CMake infrastructure that enables externally maintained addons/plugins to be *optionally* built with SUNDIALS. See [Contributing](#) for details.

### New Features and Enhancements

Added support for Kokkos Kernels v4.

Added the following Runge-Kutta Butcher tables

- ARKODE\_FORWARD\_EULER\_1\_1
- ARKODE\_RALSTON\_EULER\_2\_1\_2
- ARKODE\_EXPLICIT\_MIDPOINT\_EULER\_2\_1\_2
- ARKODE\_BACKWARD\_EULER\_1\_1
- ARKODE\_IMPLICIT\_MIDPOINT\_1\_2
- ARKODE\_IMPLICIT\_TRAPEZOIDAL\_2\_2

Added the following MRI coupling tables

- ARKODE\_MRI\_GARK\_FORWARD\_EULER
- ARKODE\_MRI\_GARK\_RALSTON2
- ARKODE\_MRI\_GARK\_RALSTON3
- ARKODE\_MRI\_GARK\_BACKWARD\_EULER
- ARKODE\_MRI\_GARK\_IMPLICIT\_MIDPOINT
- ARKODE\_IMEX\_MRI\_GARK\_EULER
- ARKODE\_IMEX\_MRI\_GARK\_TRAPEZOIDAL
- ARKODE\_IMEX\_MRI\_GARK\_MIDPOINT



Added `ARKodeButcherTable_ERKIDToName()` and `ARKodeButcherTable_DIRKIDToName()` to convert a Butcher table ID to a string representation.

Added the function `ARKodeSetAutonomous()` in ARKODE to indicate that the implicit right-hand side function does not explicitly depend on time. When using the trivial predictor, an autonomous problem may reuse implicit function evaluations across stage solves to reduce the total number of function evaluations.

Users may now disable interpolated output in ARKODE by passing `ARK_INTERP_NONE` to `ARKodeSetInterpolantType()`. When interpolation is disabled, rootfinding is not supported, implicit methods must use the trivial predictor (the default option), and interpolation at stop times cannot be used (interpolating at stop times is disabled by default). With interpolation disabled, calling `ARKodeEvolve()` in `ARK_NORMAL` mode will return at or past the requested output time (setting a stop time may still be used to halt the integrator at a specific time). Disabling interpolation will reduce the memory footprint of an integrator by two or more state vectors (depending on the interpolant type and degree) which can be beneficial when interpolation is not needed e.g., when integrating to a final time without output in between or using an explicit fast time scale integrator with an MRI method.

Added “Resize” capability to ARKODE’s `SPRKStep` time-stepping module.

Enabled the Fortran interfaces to build with 32-bit `sunindextype`.

### Bug Fixes

Updated the CMake variable `HIP_PLATFORM` default to `amd` as the previous default, `hcc`, is no longer recognized in ROCm 5.7.0 or newer. The new default is also valid in older version of ROCm (at least back to version 4.3.1).

Renamed the DPCPP value for the `SUNDIALS_GINKGO_BACKENDS` CMake option to `SYCL` to match Ginkgo’s updated naming convention.

Changed the CMake version compatibility mode for SUNDIALS to `AnyNewerVersion` instead of `SameMajorVersion`. This fixes the issue seen [here](#).

Fixed a CMake bug that caused an MPI linking error for our C++ examples in some instances. Fixes [GitHub Issue #464](#).

Fixed the runtime library installation path for windows systems. This fix changes the default library installation path from `CMAKE_INSTALL_PREFIX/CMAKE_INSTALL_LIBDIR` to `CMAKE_INSTALL_PREFIX/CMAKE_INSTALL_BINDIR`.

Fixed conflicting `.lib` files between shared and static libs when using `MSVC` on Windows

Fixed invalid `SUNDIALS_EXPORT` generated macro when building both shared and static libs.

Fixed a bug in some Fortran examples where `c_null_ptr` was passed as an argument to a function pointer instead of `c_null_funptr`. This caused compilation issues with the Cray Fortran compiler.

Fixed a bug in the HIP execution policies where `WARP_SIZE` would not be set with ROCm 6.0.0 or newer.

Fixed a bug that caused error messages to be cut off in some cases. Fixes [GitHub Issue #461](#).

Fixed a memory leak when an error handler was added to a `SUNContext`. Fixes [GitHub Issue #466](#).

Fixed a bug where `MRIStepEvolve()` would not handle a recoverable error produced from evolving the inner stepper.

Added missing `SetRootDirection` and `SetNoInactiveRootWarn` functions to ARKODE’s `SPRKStep` time-stepping module.

Fixed a bug in `ARKodeSPRKTable_Create()` where the coefficient arrays were not allocated.

Fix bug on LLP64 platforms (like Windows 64-bit) where `KLU_INDEXTYPE` could be 32 bits wide even if `SUNDIALS_INT64_T` is defined.

Check if size of `SuiteSparse_long` is 8 if the size of `sunindextype` is 8 when using `KLU`.

Fixed several build errors with the Fortran interfaces on Windows systems.

### Deprecation Notices

Numerous ARKODE stepper-specific functions are now deprecated in favor of ARKODE-wide functions.

Deprecated the *ARKStepSetOptimalParams* function. Since this function does not have an ARKODE-wide equivalent, instructions have been added to the user guide for how to retain the current functionality using other user-callable functions.

The unsupported implementations of *N\_VGetArrayPointer* and *N\_VSetArrayPointer* for the *hypr* and PETSc vectors are now deprecated. Users should access the underlying wrapped external library vector objects instead with *N\_VGetVector\_ParHyp* and *N\_VGetVector\_Petsc*, respectively.

## 23.10 Changes to SUNDIALS in release 7.0.0

### Major Feature

SUNDIALS now has more robust and uniform error handling. Non-release builds will be built with additional error checking by default. See §4.3 for details.

### Breaking Changes

#### *Minimum C Standard*

SUNDIALS now requires using a compiler that supports a subset of the C99 standard. Note with the Microsoft C/C++ compiler the subset of C99 features utilized by SUNDIALS are available starting with [Visual Studio 2015](#).

#### *Minimum CMake Version*

CMake 3.18 or newer is now required when building SUNDIALS.

#### *Deprecated Types and Functions Removed*

The previously deprecated types *realtype* and *booleantype* were removed from *sundials\_types.h* and replaced with *sunrealtype* and *sunbooleantype*. The deprecated names for these types can be used by including the header file *sundials\_types\_deprecated.h* but will be fully removed in the next major release. Functions, types and header files that were previously deprecated have also been removed.

#### *Error Handling Changes*

With the addition of the new error handling capability, the *\*SetErrHandlerFn* and *\*SetErrFile* functions in CVODE(S), IDA(S), ARKODE, and KINSOL have been removed. Users of these functions can use the functions *SUNContext\_PushErrHandler()*, and *SUNLogger\_SetErrorFilename()* instead. For further details see Sections §4.3 and §4.4.

In addition the following names/symbols were replaced by *SUN\_ERR\_\** codes:

Removed	Replaced with SUNErrCode
SUNLS_SUCCESS	SUN_SUCCESS
SUNLS_UNRECOV_FAILURE	no replacement (value was unused)
SUNLS_MEM_NULL	SUN_ERR_ARG_CORRUPT
SUNLS_ILL_INPUT	SUN_ERR_ARG_*
SUNLS_MEM_FAIL	SUN_ERR_MEM_FAIL
SUNLS_PACKAGE_FAIL_UNREC	SUN_ERR_EXT_FAIL
SUNLS_VECTOROP_ERR	SUN_ERR_OP_FAIL
SUN-NLS_SUCCESS	SUN_SUCCESS
SUN-NLS_MEM_NULL	SUN_ERR_ARG_CORRUPT
SUN-NLS_MEM_FAIL	SUN_ERR_MEM_FAIL
SUN-NLS_ILL_INPUT	SUN_ERR_ARG_*
SUN-NLS_VECTOROP_ERR	SUN_ERR_OP_FAIL
SUN-NLS_EXT_FAIL	SUN_ERR_EXT_FAIL
SUNMAT_SUCCESS	SUN_SUCCESS
SUNMAT_ILL_INPUT	SUN_ERR_ARG_*
SUNMAT_MEM_FAIL	SUN_ERR_MEM_FAIL
SUNMAT_OPERATION_FAIL	SUN_ERR_OP_FAIL
SUNMAT_MATVEC_SETUP_REQUIRED	SUN_ERR_OP_FAIL

The following functions have had their signature updated to ensure they can leverage the new SUNDIALS error handling capabilities.

- From `sundials_futils.h`
  - `SUNDIALSFileOpen()`
  - `SUNDIALSFileClose()`
- From `sundials_memory.h`
  - `SUNMemoryNewEmpty()`
  - `SUNMemoryHelper_Alias()`
  - `SUNMemoryHelper_Wrap()`
- From `sundials_nvector.h`
  - `N_VNewVectorArray()`

#### *SUNComm Type Added*

We have replaced the use of a type-erased (i.e., `void*`) pointer to a communicator in place of `MPI_Comm` throughout the SUNDIALS API with a `SUNComm`, which is just a typedef to an `int` in builds without MPI and a typedef to a `MPI_Comm` in builds with MPI. As a result:

- When MPI is enabled, all SUNDIALS libraries will include MPI symbols and applications will need to include the path for MPI headers and link against the corresponding MPI library.
- All users will need to update their codes because the call to `SUNContext_Create()` now takes a `SUNComm` instead of type-erased pointer to a communicator. For non-MPI codes, pass `SUN_COMM_NULL` to the `comm` argument instead of `NULL`. For MPI codes, pass the `MPI_Comm` directly.
- The same change must be made for calls to `SUNLogger_Create()` or `SUNProfiler_Create()`.
- Some users will need to update their calls to `N_VGetCommunicator()`, and update any custom `N_Vector` implementations that provide `N_VGetCommunicator()`, since it now returns a `SUNComm`.

The change away from type-erased pointers for [SUNComm](#) fixes problems like the one described in [GitHub Issue #275](#).

The SUNLogger is now always MPI-aware if MPI is enabled in SUNDIALS and the SUNDIALS\_LOGGING\_ENABLE\_MPI CMake option and macro definition were removed accordingly.

#### *SUNDIALS Core Library*

Users now need to link to `sundials_core` in addition to the libraries already linked to. This will be picked up automatically in projects that use the SUNDIALS CMake target. The library `sundials_generic` has been superseded by `sundials_core` and is no longer available. This fixes some duplicate symbol errors on Windows when linking to multiple SUNDIALS libraries.

#### *Fortran Interface Modules Streamlined*

We have streamlined the Fortran modules that need to be included by users by combining the SUNDIALS core into one Fortran module, `fsundials_core_mod`. Modules for implementations of the core APIs still exist (e.g., for the Dense linear solver there is `fsunlinsol_dense_mod`) as do the modules for the SUNDIALS packages (e.g., `fcvode_mod`). The following modules are the ones that have been consolidated into `fsundials_core_mod`:

```
fsundials_adaptcontroller_mod
fsundials_context_mod
fsundials_futils_mod
fsundials_linearsolver_mod
fsundials_logger_mod
fsundials_matrix_mod
fsundials_nonlinearsolver_mod
fsundials_nvector_mod
fsundials_profiler_mod
fsundials_types_mod
```

#### **Minor Changes**

The `CMAKE_BUILD_TYPE` defaults to `RelWithDebInfo` mode now i.e., SUNDIALS will be built with optimizations and debugging symbols enabled by default. Previously the build type was unset by default so no optimization or debugging flags were set.

The advanced CMake options to override the inferred LAPACK name-mangling scheme have been updated from `SUNDIALS_F77_FUNC_CASE` and `SUNDIALS_F77_FUNC_UNDERSCORES` to [SUNDIALS\\_LAPACK\\_CASE](#) and [SUNDIALS\\_LAPACK\\_UNDERSCORES](#), respectively.

As a subset of C99 is now required the CMake option `USE_GENERIC_MATH` as been removed.

The C++ convenience classes (e.g., `sundials::Context`) have been moved to from SUNDIALS `.h` headers to corresponding `.hpp` headers (e.g., `sundials/sundials_context.hpp`) so C++ codes do not need to compile with C++14 support when using the C API.

Converted most previous Fortran 77 and 90 examples to use SUNDIALS' Fortran 2003 interface.

#### **Bug Fixes**

Fixed [GitHub Issue #329](#) so that C++20 aggregate initialization can be used.

Fixed integer overflow in the internal SUNDIALS hashmap. This resolves [GitHub Issues #409](#) and [#249](#).

#### **Deprecation Notice**

The functions in `sundials_math.h` will be deprecated in the next release.

```
sunrealtype SUNRpowerI(sunrealtype base, int exponent);
sunrealtype SUNRpowerR(sunrealtype base, sunrealtype exponent);
sunbooleantype SUNRCompare(sunrealtype a, sunrealtype b);
```

(continues on next page)

(continued from previous page)

```
sunboolean_t SUNRCompareTol(sunreal_t a, sunreal_t b, sunreal_t tol);
sunreal_t SUNStrToReal(const char* str);
```

Additionally, the following header files (and everything in them) will be deprecated – users who rely on these are recommended to transition to the corresponding *SUNMatrix* and *SUNLinearSolver* modules:

```
sundials_direct.h
sundials_dense.h
sundials_band.h
```

## 23.11 Changes to SUNDIALS in release 6.7.0

### Major Feature

Added the *SUNAdaptController* base class, ported ARKODE's internal implementations of time step controllers to implementations of this class, and updated ARKODE to use these objects instead of its own implementations. Added *ARKStepSetAdaptController()* and *ERKStepSetAdaptController()* routines so that users can modify controller parameters, or even provide custom implementations.

### New Features

Improved the computational complexity of the sparse matrix *ScaleAddI* function from  $\mathcal{O}(M * N)$  to  $\mathcal{O}(\text{NNZ})$ .

Added Fortran support for the LAPACK dense linear solver implementation.

Added the routines *ARKStepSetAdaptivityAdjustment()* and *ERKStepSetAdaptivityAdjustment()*, that allow users to adjust the value for the method order supplied to the temporal adaptivity controllers. The ARKODE default for this adjustment has been  $-1$  since its initial release, but for some applications a value of  $0$  is more appropriate. Users who notice that their simulations encounter a large number of temporal error test failures may want to experiment with adjusting this value.

Added the third order ERK method *ARKODE\_SHU\_OSHER\_3\_2\_3*, the fourth order ERK method *ARKODE\_SOFRONIOU-SPALETTA\_5\_3\_4*, the sixth order ERK method *ARKODE\_VERNER\_9\_5\_6*, the seventh order ERK method *ARKODE-VERNER\_10\_6\_7*, the eighth order ERK method *ARKODE\_VERNER\_13\_7\_8*, and the ninth order ERK method *ARKODE-VERNER\_16\_8\_9*.

*ARKStep*, *ERKStep*, *MRISStep*, and *SPRKStep* were updated to remove a potentially unnecessary right-hand side evaluation at the end of an integration. *ARKStep* was additionally updated to remove extra right-hand side evaluations when using an explicit method or an implicit method with an explicit first stage.

The *MRISStepInnerStepper* class in *MRISStep* was updated to make supplying an *MRISStepInnerFullRhsFn* optional.

### Bug Fixes

Changed the *SUNProfiler* so that it does not rely on *MPI\_WTime* in any case. This fixes [GitHub Issue #312](#).

Fixed scaling bug in *SUNMatScaleAddI\_Sparse* for non-square matrices.

Fixed a regression introduced by the stop time bug fix in v6.6.1 where ARKODE, CVODE, CVODES, IDA, and IDAS would return at the stop time rather than the requested output time if the stop time was reached in the same step in which the output time was passed.

Fixed a bug in *ERKStep* where methods with  $c_s = 1$  but  $a_{s,j} \neq b_j$  were incorrectly treated as having the first same as last (FSAL) property.

Fixed a bug in ARKODE where *ARKStepSetInterpolateStopTime()* would return an interpolated solution at the stop time in some cases when interpolation was disabled.

Fixed a bug in [ARKStepSetTableNum\(\)](#) wherein it did not recognize `ARKODE_ARK2_ERK_3_1_2` and `ARKODE_ARK2_DIRK_3_1_2` as a valid additive Runge–Kutta Butcher table pair.

Fixed a bug in [MRIStepCoupling\\_Write\(\)](#) where explicit coupling tables were not written to the output file pointer.

Fixed missing soversions in some [SUNLinearSolver](#) and [SUNNonlinearSolver](#) CMake targets.

Renamed some internal types in CVODES and IDAS to allow both packages to be built together in the same binary.

## 23.12 Changes to SUNDIALS in release 6.6.2

Fixed the build system support for MAGMA when using a NVIDIA HPC SDK installation of CUDA and fixed the targets used for rocBLAS and rocSPARSE.

## 23.13 Changes to SUNDIALS in release 6.6.1

### New Features

Updated the Trilinos Tpetra [N\\_Vector](#) interface to support Trilinos 14.

### Bug Fixes

Fixed a memory leak when destroying a CUDA, HIP, SYCL, or system [SUNMemoryHelper](#) object.

Fixed a bug in ARKODE, CVODE, CVODES, IDA, and IDAS where the stop time may not be cleared when using normal mode if the requested output time is the same as the stop time. Additionally, with ARKODE, CVODE, and CVODES this fix removes an unnecessary interpolation of the solution at the stop time that could occur in this case.

## 23.14 Changes to SUNDIALS in release 6.6.0

### Major Features

A new time-stepping module, [SPRKStep](#), was added to ARKODE. This time-stepper provides explicit symplectic partitioned Runge–Kutta methods up to order 10 for separable Hamiltonian systems.

Added support for relaxation Runge–Kutta methods in ERKStep and ARKStep, see [Relaxation Methods](#), [Relaxation Methods](#), and [Relaxation Methods](#) for more information.

### New Features

Updated the default ARKODE, CVODE, and CVODES behavior when returning the solution when the internal time has reached a user-specified stop time. Previously, the output solution was interpolated to the value of `tstop`; the default is now to copy the internal solution vector. Users who wish to revert to interpolation may call a new routine [CvodeSetInterpolateStopTime\(\)](#), [ARKStepSetInterpolateStopTime\(\)](#), [ERKStepSetInterpolateStopTime\(\)](#), or [MRIStepSetInterpolateStopTime\(\)](#).

Added the second order IMEX method from [50] as the default second order IMEX method in ARKStep. The explicit table is given by `ARKODE_ARK2_ERK_3_1_2` and the implicit table by `ARKODE_ARK2_DIRK_3_1_2`.

Updated the F2003 utility routines [SUNDIALSFileOpen\(\)](#) and [SUNDIALSFileClose\(\)](#) to support user specification of `stdout` and `stderr` strings for the output file names.

### Bug Fixes

A potential bug was fixed when using inequality constraint handling and calling [ARKStepGetEstLocalErrors\(\)](#) or [ERKStepGetEstLocalErrors\(\)](#) after a failed step in which an inequality constraint violation occurred. In this case, the values returned by [ARKStepGetEstLocalErrors\(\)](#) or [ERKStepGetEstLocalErrors\(\)](#) may have been invalid.

## 23.15 Changes to SUNDIALS in release 6.5.1

### New Features

Added the following functions to disable a previously set stop time:

- [\*ARKStepClearStopTime\(\)\*](#)
- [\*ERKStepClearStopTime\(\)\*](#)
- [\*MRISStepClearStopTime\(\)\*](#)
- [\*CVodeClearStopTime\(\)\*](#)
- [\*IDAClearStopTime\(\)\*](#)

The default interpolant in ARKODE when using a first order method has been updated to a linear interpolant to ensure values obtained by the integrator are returned at the ends of the time interval. To restore the previous behavior of using a constant interpolant call [\*ARKStepSetInterpolantDegree\(\)\*](#), [\*ERKStepSetInterpolantDegree\(\)\*](#), or [\*MRISStepSetInterpolantDegree\(\)\*](#) and set the interpolant degree to zero before evolving the problem.

### Bug Fixes

Fixed build errors when using SuperLU\_DIST with ROCM enabled to target AMD GPUs.

Fixed compilation errors in some SYCL examples when using the `icx` compiler.

## 23.16 Changes to SUNDIALS in release 6.5.0

### New Features

A new capability to keep track of memory allocations made through the [\*SUNMemoryHelper\*](#) classes has been added. Memory allocation stats can be accessed through the [\*SUNMemoryHelper\\_GetAllocStats\(\)\*](#) function. See §16.1 for more details.

Added the following functions to assist in debugging simulations utilizing matrix-based linear solvers:

- [\*ARKStepGetJac\(\)\*](#)
- [\*ARKStepGetJacTime\(\)\*](#)
- [\*ARKStepGetJacNumSteps\(\)\*](#)
- [\*MRISStepGetJac\(\)\*](#)
- [\*MRISStepGetJacTime\(\)\*](#)
- [\*MRISStepGetJacNumSteps\(\)\*](#)
- [\*CVodeGetJac\(\)\*](#)
- [\*CVodeGetJacTime\(\)\*](#)
- [\*CVodeGetJacNumSteps\(\)\*](#)
- [\*IDAGetJac\(\)\*](#)
- [\*IDAGetJacCj\(\)\*](#)
- [\*IDAGetJacTime\(\)\*](#)
- [\*IDAGetJacNumSteps\(\)\*](#)
- [\*KINGetJac\(\)\*](#)



- [KINGGetJacNumIters\(\)](#)

Added support for CUDA 12.

Added support for the SYCL backend with RAJA 2022.x.y.

### Bug Fixes

Fixed an underflow bug during root finding in ARKODE, CVODE, CVODES, IDA and IDAS. This fixes [GitHub Issue #57](#).

Fixed an issue with finding oneMKL when using the `icpx` compiler with the `-fsycl` flag as the C++ compiler instead of `dpcpp`.

Fixed the shape of the arrays returned by the Fortran interfaces to [N\\_VGetArrayPointer\(\)](#), [SUNDenseMatrix\\_Data\(\)](#), [SUNBandMatrix\\_Data\(\)](#), [SUNSparseMatrix\\_Data\(\)](#), [SUNSparseMatrix\\_IndexValues\(\)](#), and [SUNSparseMatrix\\_IndexPointers\(\)](#). Compiling and running code that uses the SUNDIALS Fortran interfaces with bounds checking will now work.

Fixed an implicit conversion error in the Butcher table for ESDIRK5(4)7L[2]SA2.

## 23.17 Changes to SUNDIALS in release 6.4.1

Fixed a bug with the Kokkos interfaces that would arise when using clang.

Fixed a compilation error with the Intel oneAPI 2022.2 Fortran compiler in the Fortran 2003 interface test for the serial [N\\_Vector](#).

Fixed a bug in the LAPACK band and dense linear solvers which would cause the tests to fail on some platforms.

## 23.18 Changes to SUNDIALS in release 6.4.0

### New Requirements

CMake 3.18.0 or newer is now required for CUDA support.

A C++14 compliant compiler is now required for C++ based features and examples e.g., CUDA, HIP, RAJA, Trilinos, SuperLU\_DIST, MAGMA, Ginkgo, and Kokkos.

### Major Features

Added support for the [Ginkgo](#) linear algebra library. This support includes new SUNDIALS matrix and linear solver implementations, see the sections [§9.10](#) and [§10.18](#).

Added new SUNDIALS vector, dense matrix, and dense linear solver implementations utilizing the [Kokkos Ecosystem](#) for performance portability, see sections [§8.14](#), [§9.12](#), and [§10.20](#) for more information.

### New Features

Added support for GPU enabled SuperLU\_DIST and SuperLU\_DIST v8.x.x. Removed support for SuperLU\_DIST v6.x.x or older. Fix mismatched definition and declaration bug in SuperLU\_DIST matrix constructor.

Added the functions following functions to load a Butcher table from a string:

- [ARKStepSetTableName\(\)](#)
- [ERKStepSetTableName\(\)](#)
- [MRISStepCoupling\\_LoadTableByName\(\)](#)
- [ARKodeButcherTable\\_LoadDIRKByName\(\)](#)



- [`ARKodeButcherTable\_LoadERKByName\(\)`](#)

### Bug Fixes

Fixed a bug in the CUDA and HIP vectors where [`N\_VMaxNorm\(\)`](#) would return the minimum positive floating-point value for the zero vector.

Fixed memory leaks/out of bounds memory accesses in the ARKODE MRISep module that could occur when attaching a coupling table after reinitialization with a different number of stages than originally selected.

Fixed a memory leak where the projection memory would not be deallocated when calling [`CVodeFree\(\)`](#).

## 23.19 Changes to SUNDIALS in release 6.3.0

### New Features

Added the following functions to retrieve the user data pointer provided with `SetUserData` functions:

- [`ARKStepGetUserData\(\)`](#)
- [`ERKStepGetUserData\(\)`](#)
- [`MRISepGetUserData\(\)`](#)
- [`CVodeGetUserData\(\)`](#)
- [`IDAGetUserData\(\)`](#)
- [`KINGetUserData\(\)`](#)

Added a variety of embedded DIRK methods from [69] and [70].

Updated [`MRISepReset\(\)`](#) to call the corresponding [`MRISepInnerResetFn`](#) with the same `tR` and `yR` arguments for the [`MRISepInnerStepper`](#) object that is used to evolve the MRI “fast” time scale subproblems.

Added a new example (`examples/cvode/serial/cvRocket_dns.c`) which demonstrates using CVODE with a discontinuous right-hand-side function and rootfinding.

### Bug Fixes

Fixed a bug in [`ERKStepReset\(\)`](#), [`ERKStepReInit\(\)`](#), [`ARKStepReset\(\)`](#), [`ARKStepReInit\(\)`](#), [`MRISepReset\(\)`](#), and [`MRISepReInit\(\)`](#) where a previously-set value of `tstop` (from a call to [`ERKStepSetStopTime\(\)`](#), [`ARKStepSetStopTime\(\)`](#), or [`MRISepSetStopTime\(\)`](#), respectively) would not be cleared.

Fixed the unintuitive behavior of the `USE_GENERIC_MATH` CMake option which caused the double precision math functions to be used regardless of the value of [`SUNDIALS\_PRECISION`](#). Now, SUNDIALS will use precision appropriate math functions when they are available and the user may provide the math library to link to via the advanced CMake option [`SUNDIALS\_MATH\_LIBRARY`](#).

Changed `SUNDIALS_LOGGING_ENABLE_MPI` CMake option default to be OFF. This fixes [GitHub Issue #177](#).

## 23.20 Changes to SUNDIALS in release 6.2.0

### Major Features

Added the [`SUNLogger`](#) API which provides a SUNDIALS-wide mechanism for logging of errors, warnings, informational output, and debugging output.

Added support to CVODES for integrating IVPs with constraints using BDF methods and projecting the solution onto the constraint manifold with a user defined projection function. This implementation is accompanied by additions to the CVODES user documentation and examples.

## New Features

Added the function `SUNProfiler_Reset()` to reset the region timings and counters to zero.

Added the following functions to output all of the integrator, nonlinear solver, linear solver, and other statistics in one call:

- `ARKStepPrintAllStats()`
- `ERKStepPrintAllStats()`
- `MRIStepPrintAllStats()`
- `CNodePrintAllStats()`
- `IDAPrintAllStats()`
- `KINPrintAllStats()`

The file `scripts/sundials_csv.py` contains functions for parsing the comma-separated value (CSV) output files when using the CSV output format.

Added functions to CVODE, CVODES, IDA, and IDAS to change the default step size adaptivity parameters. For more information see the documentation for:

- `CNodeSetEtaFixedStepBounds()`
- `CNodeSetEtaMaxFirstStep()`
- `CNodeSetEtaMaxEarlyStep()`
- `CNodeSetNumStepsEtaMaxEarlyStep()`
- `CNodeSetEtaMax()`
- `CNodeSetEtaMin()`
- `CNodeSetEtaMinErrFail()`
- `CNodeSetEtaMaxErrFail()`
- `CNodeSetNumFailsEtaMaxErrFail()`
- `CNodeSetEtaConvFail()`
- `IDASetEtaFixedStepBounds()`
- `IDASetEtaMax()`
- `IDASetEtaMin()`
- `IDASetEtaLow()`
- `IDASetEtaMinErrFail()`
- `IDASetEtaConvFail()`

Added the functions `ARKStepSetDeduceImplicitRhs()` and `MRIStepSetDeduceImplicitRhs()` to optionally remove an evaluation of the implicit right-hand side function after nonlinear solves. See *Nonlinear solver methods*, for considerations on using this optimization.

Added the function `MRIStepSetOrder()` to select the default MRI method of a given order.

Added the functions `CNodeSetDeltaGammaMaxLSetup()` and `CNodeSetDeltaGammaMaxBadJac()` in CVODE and CVODES to adjust the  $\gamma$  change thresholds to require a linear solver setup or Jacobian/precondition update, respectively.

Added the function `IDASetDeltaCjLSetup()` in IDA and IDAS to adjust the parameter that determines when a change in  $c_j$  requires calling the linear solver setup function.

Added the function `IDASSetMinStep()` to set a minimum step size.

### Bug Fixes

Fixed the `SUNContext` convenience class for C++ users to disallow copy construction and allow move construction.

The behavior of `N_VSetKernelExecPolicy_Sycl()` has been updated to be consistent with the CUDA and HIP vectors. The input execution policies are now cloned and may be freed after calling `N_VSetKernelExecPolicy_Sycl()`. Additionally, NULL inputs are now allowed and, if provided, will reset the vector execution policies to the defaults.

A memory leak in the SYCL vector was fixed where the execution policies were not freed when the vector was destroyed.

The include guard in `nvector_mpmmanyvector.h` has been corrected to enable using both the ManyVector and MPI-ManyVector vector implementations in the same simulation.

A bug was fixed in the ARKODE, CVODE(S), and IDA(S) functions to retrieve the number of nonlinear solver failures. The failure count returned was the number of failed *steps* due to a nonlinear solver failure i.e., if a nonlinear solve failed with a stale Jacobian or preconditioner but succeeded after updating the Jacobian or preconditioner, the initial failure was not included in the nonlinear solver failure count. The following functions have been updated to return the total number of nonlinear solver failures:

- `ARKStepGetNumNonlinSolvConvFails()`
- `ARKStepGetNonlinSolvStats()`
- `MRIStepGetNumNonlinSolvConvFails()`
- `MRIStepGetNonlinSolvStats()`
- `CVodeGetNumNonlinSolvConvFails()`
- `CVodeGetNonlinSolvStats()`
- `CVodeGetSensNumNonlinSolvConvFails()`
- `CVodeGetSensNonlinSolvStats()`
- `CVodeGetStgrSensNumNonlinSolvConvFails()`
- `CVodeGetStgrSensNonlinSolvStats()`
- `IDAGetNumNonlinSolvConvFails()`
- `IDAGetNonlinSolvStats()`
- `IDAGetSensNumNonlinSolvConvFails()`
- `IDAGetSensNonlinSolvStats()`

As a result of this change users may see an increase in the number of failures reported from the above functions. The following functions have been added to retrieve the number of failed steps due to a nonlinear solver failure i.e., the counts previously returned by the above functions:

- `ARKStepGetNumStepSolveFails()`
- `MRIStepGetNumStepSolveFails()`
- `CVodeGetNumStepSolveFails()`
- `CVodeGetNumStepSensSolveFails()`
- `CVodeGetNumStepStgrSensSolveFails()`
- `IDAGetNumStepSolveFails()`
- `IDAGetNumStepSensSolveFails()`

Changed exported SUNDIALS PETSc CMake targets to be INTERFACE IMPORTED instead of UNKNOWN IMPORTED.

### Deprecation Notice

Deprecated the following functions, it is recommended to use the [SUNLogger](#) API instead.

- ARKStepSetDiagnostics
- ERKStepSetDiagnostics
- MRISetSetDiagnostics
- KINSetInfoFile
- SUNNonlinSolSetPrintLevel\_Newton
- SUNNonlinSolSetInfoFile\_Newton
- SUNNonlinSolSetPrintLevel\_FixedPoint
- SUNNonlinSolSetInfoFile\_FixedPoint
- SUNLinSolSetInfoFile\_PCG
- SUNLinSolSetPrintLevel\_PCG
- SUNLinSolSetInfoFile\_SPGMR
- SUNLinSolSetPrintLevel\_SPGMR
- SUNLinSolSetInfoFile\_SPFQMR
- SUNLinSolSetPrintLevel\_SPFQMR
- SUNLinSolSetInfoFile\_SPTFQM
- SUNLinSolSetPrintLevel\_SPTFQMR
- SUNLinSolSetInfoFile\_SPBCGS
- SUNLinSolSetPrintLevel\_SPBCGS

The `SUNLinSolSetInfoFile_*` and `SUNNonlinSolSetInfoFile_*` family of functions are now enabled by setting the CMake option [SUNDIALS\\_LOGGING\\_LEVEL](#) to a value  $\geq 3$ .

## 23.21 Changes to SUNDIALS in release 6.1.1

### New Feature

Added new Fortran example program, `examples/arkode/F2003_serial/ark_kpr_mri_f2003.f90` demonstrating MRI capabilities.

### Bug Fixes

Fixed exported `SUNDIALSConfig.cmake`.

Fixed Fortran interface to [MRIStepInnerStepper](#) and [MRIStepCoupling](#) structures and functions.

## 23.22 Changes to SUNDIALS in release 6.1.0

### New Features

Added new reduction implementations for the CUDA and HIP vectors that use shared memory (local data storage) instead of atomics. These new implementations are recommended when the target hardware does not provide atomic support for the floating point precision that SUNDIALS is being built with. The HIP vector uses these by default, but the `N_VSetKernelExecPolicy_Cuda()` and `N_VSetKernelExecPolicy_Hip()` functions can be used to choose between different reduction implementations.

SUNDIALS: <lib> targets with no static/shared suffix have been added for use within the build directory (this mirrors the targets exported on installation).

`CMAKE_C_STANDARD` is now set to 99 by default.

### Bug Fixes

Fixed exported SUNDIALSConfig.cmake when profiling is enabled without Caliper.

Fixed sundials\_export.h include in sundials\_config.h.

Fixed memory leaks in the SuperLU\_MT linear solver interface.

## 23.23 Changes to SUNDIALS in release 6.0.0

### Breaking Changes

#### *SUNContext Object Added*

SUNDIALS v6.0.0 introduces a new `SUNContext` object on which all other SUNDIALS objects depend. As such, the constructors for all SUNDIALS packages, vectors, matrices, linear solvers, nonlinear solvers, and memory helpers have been updated to accept a context as the last input. Users upgrading to SUNDIALS v6.0.0 will need to call `SUNContext_Create()` to create a context object with before calling any other SUNDIALS library function, and then provide this object to other SUNDIALS constructors. The context object has been introduced to allow SUNDIALS to provide new features, such as the profiling/instrumentation also introduced in this release, while maintaining thread-safety. See the §4.2 for more details.

The script `scripts/upgrade-to-sundials-6-from-5.sh` has been provided with this release (and obtainable from the GitHub release page) to help ease the transition to SUNDIALS v6.0.0. The script will add a `SUNCTX_PLACEHOLDER` argument to all of the calls to SUNDIALS constructors that now require a `SUNContext` object. It can also update deprecated SUNDIALS constants/types to the new names. It can be run like this:

```
./upgrade-to-sundials-6-from-5.sh <files to update>
```

#### *Updated SUNMemoryHelper Function Signatures*

The `SUNMemoryHelper` functions `SUNMemoryHelper_Alloc()`, `SUNMemoryHelper_Dealloc()`, and `SUNMemoryHelper_Copy()` have been updated to accept an opaque handle as the last input. At a minimum, user-defined `SUNMemoryHelper` implementations will need to update these functions to accept the additional argument. Typically, this handle is the execution stream (e.g., a CUDA/HIP stream or SYCL queue) for the operation. The CUDA, HIP, and SYCL implementations have been updated accordingly. Additionally, the constructor `SUNMemoryHelper_Sycl()` has been updated to remove the SYCL queue as an input.

#### *Deprecated Functions Removed*

The previously deprecated constructor `N_VMakeWithManagedAllocator_Cuda` and the function `N_VSetCudaStream_Cuda` have been removed and replaced with `N_VNewWithMemHelp_Cuda()` and `N_VSetKernelExecPolicy_Cuda()` respectively.

The previously deprecated macros `PVEC_REAL_MPI_TYPE` and `PVEC_INTEGER_MPI_TYPE` have been removed and replaced with `MPI_SUNREALTYPE` and `MPI_SUNINDEXTYPE` respectively.

The following previously deprecated *SUNLinearSolver* functions have been removed:

Removed	Replacement
<code>SUNBandLinearSolver</code>	<code>SUNLinSol_Band()</code>
<code>SUNDenseLinearSolver</code>	<code>SUNLinSol_Dense()</code>
<code>SUNKLU</code>	<code>SUNLinSol_KLU()</code>
<code>SUNKLUReInit</code>	<code>SUNLinSol_KLUReInit()</code>
<code>SUNKLUSetOrdering</code>	<code>SUNLinSol_KLUSetOrdering()</code>
<code>SUNLapackBand</code>	<code>SUNLinSol_LapackBand()</code>
<code>SUNLapackDense</code>	<code>SUNLinSol_LapackDense()</code>
<code>SUNPCG</code>	<code>SUNLinSol_PCG()</code>
<code>SUNPCGSetPrecType</code>	<code>SUNLinSol_PCGSetPrecType()</code>
<code>SUNPCGSetMaxl</code>	<code>SUNLinSol_PCGSetMaxl()</code>
<code>SUNSPBCGS</code>	<code>SUNLinSol_SPBCGS()</code>
<code>SUNSPBCGSSetPrecType</code>	<code>SUNLinSol_SPBCGSSetPrecType()</code>
<code>SUNSPBCGSSetMaxl</code>	<code>SUNLinSol_SPBCGSSetMaxl()</code>
<code>SUNSPFGMR</code>	<code>SUNLinSol_SPFGMR()</code>
<code>SUNSPFGMRSetPrecType</code>	<code>SUNLinSol_SPFGMRSetPrecType()</code>
<code>SUNSPFGMRSetGStype</code>	<code>SUNLinSol_SPFGMRSetGStype()</code>
<code>SUNSPFGMRSetMaxRestarts</code>	<code>SUNLinSol_SPFGMRSetMaxRestarts()</code>
<code>SUNSPGMR</code>	<code>SUNLinSol_SPGMR()</code>
<code>SUNSPGMRSetPrecType</code>	<code>SUNLinSol_SPGMRSetPrecType()</code>
<code>SUNSPGMRSetGStype</code>	<code>SUNLinSol_SPGMRSetGStype()</code>
<code>SUNSPGMRSetMaxRestarts</code>	<code>SUNLinSol_SPGMRSetMaxRestarts()</code>
<code>SUNSPTFQMR</code>	<code>SUNLinSol_SPTFQMR()</code>
<code>SUNSPTFQMRSetPrecType</code>	<code>SUNLinSol_SPTFQMRSetPrecType()</code>
<code>SUNSPTFQMRSetMaxl</code>	<code>SUNLinSol_SPTFQMRSetMaxl()</code>
<code>SUNSuperLUMT</code>	<code>SUNLinSol_SuperLUMT()</code>
<code>SUNSuperLUMTSetOrdering</code>	<code>SUNLinSol_SuperLUMTSetOrdering()</code>

The deprecated functions `MRISetGetCurrentButcherTables` and `MRISetWriteButcher` and the utility functions `MRISetSetTable` and `MRISetSetTableNum` have been removed. Users wishing to create an MRI-GARK method from a Butcher table should use `MRISetCoupling_MISetMRI()` to create the corresponding MRI coupling table and attach it with `MRISetSetCoupling()`.

The previously deprecated functions `ARKSetSetMaxStepsBetweenLSet` and `ARKSetSetMaxStepsBetweenJac` have been removed and replaced with `ARKSetSetLSetupFrequency()` and `ARKSetSetJacEvalFrequency()` respectively.

The previously deprecated function `CVSetSetMaxStepsBetweenJac` has been removed and replaced with `CVSetSetJacEvalFrequency()`.

The ARKODE, CVODE, IDA, and KINSOL Fortran 77 interfaces has been removed. See §20 and the F2003 example programs for more details using the SUNDIALS Fortran 2003 module interfaces.

### Namespace Changes

The CUDA, HIP, and SYCL execution policies have been moved from the `sundials` namespace to the `sundials::cuda`, `sundials::hip`, and `sundials::sycl` namespaces respectively. Accordingly, the prefixes “Cuda”, “Hip”, and “Sycl” have been removed from the execution policy classes and methods.

The `Sundials` namespace used by the Trilinos Tpetra *N\_Vector* implementation has been replaced with the `sundials::trilinos::nvector_tpetra` namespace.

## Major Features

### Profiling Capability

A capability to profile/instrument SUNDIALS library code has been added. This can be enabled with the CMake option `SUNDIALS_BUILD_WITH_PROFILING`. A built-in profiler will be used by default, but the [Caliper](#) library can also be used instead with the CMake option `ENABLE_CALIPER`. See the documentation section on profiling for more details.

#### Warning

Profiling will impact performance, and should be enabled judiciously.

### IMEX MRI Methods and MRISetInnerStepper Object

The MRISet module has been extended to support implicit-explicit (ImEx) multirate infinitesimal generalized additive Runge–Kutta (MRI-GARK) methods. As such, [MRISetCreate\(\)](#) has been updated to include arguments for the slow explicit and slow implicit ODE right-hand side functions. [MRISetCreate\(\)](#) has also been updated to require attaching an MRISetInnerStepper for evolving the fast time scale. [MRISetReInit\(\)](#) has been similarly updated to take explicit and implicit right-hand side functions as input. Codes using explicit or implicit MRI methods will need to update [MRISetCreate\(\)](#) and [MRISetReInit\(\)](#) calls to pass NULL for either the explicit or implicit right-hand side function as appropriate. If ARKStep is used as the fast time scale integrator, codes will need to call [ARKStepCreateMRISetInnerStepper\(\)](#) to wrap the ARKStep memory as an MRISetInnerStepper object. Additionally, [MRISetGetNumRhsEvals\(\)](#) has been updated to return the number of slow implicit and explicit function evaluations. The coupling table, [MRISetCoupling](#), and the functions [MRISetCouplingAlloc\(\)](#) and [MRISetCouplingCreate\(\)](#) have also been updated to support IMEX-MRI-GARK methods.

### New Features

Two new optional vector operations, [N\\_VDotProdMultiLocal\(\)](#) and [N\\_VDotProdMultiAllReduce\(\)](#), have been added to support low-synchronization methods for Anderson acceleration.

The implementation of solve-decoupled implicit MRI-GARK methods has been updated to remove extraneous slow implicit function calls and reduce the memory requirements.

Added a new function [CvodeGetLinSolveStats\(\)](#) to get the CVODES linear solver statistics as a group.

Added a new function, [CvodeSetMonitorFn\(\)](#), that takes a user-function to be called by CVODES after every `nst` successfully completed time-steps. This is intended to provide a way of monitoring the CVODES statistics throughout the simulation.

New orthogonalization methods were added for use within the KINSOL Anderson acceleration routine. See [Anderson Acceleration QR Factorization](#) and [KINSetOrthAA\(\)](#) for more details.

### Deprecation Notice

The serial, PThreads, PETSc, *hypr*, Parallel, OpenMP\_DEV, and OpenMP vector functions [N\\_VCloneVectorArray\\_\\*](#) and [N\\_VDestroyVectorArray\\_\\*](#) have been deprecated. The generic [N\\_VCloneVectorArray\(\)](#) and [N\\_VDestroyVectorArray\(\)](#) functions should be used instead.

Many constants, types, and functions have been renamed so that they are properly namespaced. The old names have been deprecated and will be removed in SUNDIALS v7.0.0.

The following constants, macros, and typedefs are now deprecated:

Deprecated Name	New Name
<code>realtype</code>	<code>sunrealtype</code>
<code>booleantype</code>	<code>sunbooleantype</code>

continues on next page

Table 23.2 – continued from previous page

Deprecated Name	New Name
RCONST	SUN_RCONST
BIG_REAL	SUN_BIG_REAL
SMALL_REAL	SUN_SMALL_REAL
UNIT_ROUNDOFF	SUN_UNIT_ROUNDOFF
PREC_NONE	SUN_PREC_NONE
PREC_LEFT	SUN_PREC_LEFT
PREC_RIGHT	SUN_PREC_RIGHT
PREC_BOTH	SUN_PREC_BOTH
MODIFIED_GS	SUN_MODIFIED_GS
CLASSICAL_GS	SUN_CLASSICAL_GS
ATimesFn	SUNATimesFn
PSetupFn	SUNPSetupFn
PSolveFn	SUNPSolveFn
DlsMat	SUNDlsMat
DENSE_COL	SUNDLS_DENSE_COL
DENSE_ELEM	SUNDLS_DENSE_ELEM
BAND_COL	SUNDLS_BAND_COL
BAND_COL_ELEM	SUNDLS_BAND_COL_ELEM
BAND_ELEM	SUNDLS_BAND_ELEM
SDIRK_2_1_2	ARKODE_SDIRK_2_1_2
BILLINGTON_3_3_2	ARKODE_BILLINGTON_3_3_2
TRBDF2_3_3_2	ARKODE_TRBDF2_3_3_2
KVAERNO_4_2_3	ARKODE_KVAERNO_4_2_3
ARK324L2SA_DIRK_4_2_3	ARKODE_ARK324L2SA_DIRK_4_2_3
CASH_5_2_4	ARKODE_CASH_5_2_4
CASH_5_3_4	ARKODE_CASH_5_3_4
SDIRK_5_3_4	ARKODE_SDIRK_5_3_4
KVAERNO_5_3_4	ARKODE_KVAERNO_5_3_4
ARK436L2SA_DIRK_6_3_4	ARKODE_ARK436L2SA_DIRK_6_3_4
KVAERNO_7_4_5	ARKODE_KVAERNO_7_4_5
ARK548L2SA_DIRK_8_4_5	ARKODE_ARK548L2SA_DIRK_8_4_5
ARK437L2SA_DIRK_7_3_4	ARKODE_ARK437L2SA_DIRK_7_3_4
ARK548L2Sab_DIRK_8_4_5	ARKODE_ARK548L2Sab_DIRK_8_4_5
MIN_DIRK_NUM	ARKODE_MIN_DIRK_NUM
MAX_DIRK_NUM	ARKODE_MAX_DIRK_NUM
MIS_KW3	ARKODE_MIS_KW3
MRI_GARK_ERK33a	ARKODE_MRI_GARK_ERK33a
MRI_GARK_ERK45a	ARKODE_MRI_GARK_ERK45a
MRI_GARK_IRK21a	ARKODE_MRI_GARK_IRK21a
MRI_GARK_ESDIRK34a	ARKODE_MRI_GARK_ESDIRK34a
MRI_GARK_ESDIRK46a	ARKODE_MRI_GARK_ESDIRK46a
IMEX_MRI_GARK3a	ARKODE_IMEX_MRI_GARK3a
IMEX_MRI_GARK3b	ARKODE_IMEX_MRI_GARK3b
IMEX_MRI_GARK4	ARKODE_IMEX_MRI_GARK4
MIN_MRI_NUM	ARKODE_MIN_MRI_NUM
MAX_MRI_NUM	ARKODE_MAX_MRI_NUM
DEFAULT_MRI_TABLE_3	MRISTEP_DEFAULT_TABLE_3
DEFAULT_EXPL_MRI_TABLE_3	MRISTEP_DEFAULT_EXPL_TABLE_3
DEFAULT_EXPL_MRI_TABLE_4	MRISTEP_DEFAULT_EXPL_TABLE_4
DEFAULT_IMPL_SD_TABLE_2	MRISTEP_DEFAULT_IMPL_SD_TABLE_2

continues on next page



Table 23.2 – continued from previous page

Deprecated Name	New Name
DEFAULT_IMPL_SD_TABLE_3	MRISTEP_DEFAULT_IMPL_SD_TABLE_3
DEFAULT_IMPL_SD_TABLE_4	MRISTEP_DEFAULT_IMPL_SD_TABLE_4
DEFAULT_IMEX_SD_TABLE_3	MRISTEP_DEFAULT_IMEX_SD_TABLE_3
DEFAULT_IMEX_SD_TABLE_4	MRISTEP_DEFAULT_IMEX_SD_TABLE_4
HEUN_EULER_2_1_2	ARKODE_HEUN_EULER_2_1_2
BOGACKI_SHAMPINE_4_2_3	ARKODE_BOGACKI_SHAMPINE_4_2_3
ARK324L2SA_ERK_4_2_3	ARKODE_ARK324L2SA_ERK_4_2_3
ZONNEVELD_5_3_4	ARKODE_ZONNEVELD_5_3_4
ARK436L2SA_ERK_6_3_4	ARKODE_ARK436L2SA_ERK_6_3_4
SAYFY_ABURUB_6_3_4	ARKODE_SAYFY_ABURUB_6_3_4
CASH_KARP_6_4_5	ARKODE_CASH_KARP_6_4_5
FEHLBERG_6_4_5	ARKODE_FEHLBERG_6_4_5
DORMAND_PRINCE_7_4_5	ARKODE_DORMAND_PRINCE_7_4_5
ARK548L2SA_ERK_8_4_5	ARKODE_ARK548L2SA_ERK_8_4_5
VERNER_8_5_6	ARKODE_VERNER_8_5_6
FEHLBERG_13_7_8	ARKODE_FEHLBERG_13_7_8
KNOTH_WOLKE_3_3	ARKODE_KNOTH_WOLKE_3_3
ARK437L2SA_ERK_7_3_4	ARKODE_ARK437L2SA_ERK_7_3_4
ARK548L2SAb_ERK_8_4_5	ARKODE_ARK548L2SAb_ERK_8_4_5
MIN_ERK_NUM	ARKODE_MIN_ERK_NUM
MAX_ERK_NUM	ARKODE_MAX_ERK_NUM
DEFAULT_ERK_2	ARKSTEP_DEFAULT_ERK_2
DEFAULT_ERK_3	ARKSTEP_DEFAULT_ERK_3
DEFAULT_ERK_4	ARKSTEP_DEFAULT_ERK_4
DEFAULT_ERK_5	ARKSTEP_DEFAULT_ERK_5
DEFAULT_ERK_6	ARKSTEP_DEFAULT_ERK_6
DEFAULT_ERK_8	ARKSTEP_DEFAULT_ERK_8
DEFAULT_DIRK_2	ARKSTEP_DEFAULT_DIRK_2
DEFAULT_DIRK_3	ARKSTEP_DEFAULT_DIRK_3
DEFAULT_DIRK_4	ARKSTEP_DEFAULT_DIRK_4
DEFAULT_DIRK_5	ARKSTEP_DEFAULT_DIRK_5
DEFAULT_ARK_ETABLE_3	ARKSTEP_DEFAULT_ARK_ETABLE_3
DEFAULT_ARK_ETABLE_4	ARKSTEP_DEFAULT_ARK_ETABLE_4
DEFAULT_ARK_ETABLE_5	ARKSTEP_DEFAULT_ARK_ETABLE_4
DEFAULT_ARK_htable_3	ARKSTEP_DEFAULT_ARK_htable_3
DEFAULT_ARK_htable_4	ARKSTEP_DEFAULT_ARK_htable_4
DEFAULT_ARK_htable_5	ARKSTEP_DEFAULT_ARK_htable_5
DEFAULT_ERK_2	ERKSTEP_DEFAULT_2
DEFAULT_ERK_3	ERKSTEP_DEFAULT_3
DEFAULT_ERK_4	ERKSTEP_DEFAULT_4
DEFAULT_ERK_5	ERKSTEP_DEFAULT_5
DEFAULT_ERK_6	ERKSTEP_DEFAULT_6
DEFAULT_ERK_8	ERKSTEP_DEFAULT_8

In addition, the following functions are now deprecated (compile-time warnings will be printed if supported by the compiler):

Deprecated Name	New Name
DenseGETRF	SUNDlsMat_DenseGETRF

continues on next page

Table 23.3 – continued from previous page

Deprecated Name	New Name
DenseGETRS	SUNDlsMat_DenseGETRS
denseGETRF	SUNDlsMat_denseGETRF
denseGETRS	SUNDlsMat_denseGETRS
DensePOTRF	SUNDlsMat_DensePOTRF
DensePOTRS	SUNDlsMat_DensePOTRS
densePOTRF	SUNDlsMat_densePOTRF
densePOTRS	SUNDlsMat_densePOTRS
DenseGEQRF	SUNDlsMat_DenseGEQRF
DenseORMQR	SUNDlsMat_DenseORMQR
denseGEQRF	SUNDlsMat_denseGEQRF
denseORMQR	SUNDlsMat_denseORMQR
DenseCopy	SUNDlsMat_DenseCopy
denseCopy	SUNDlsMat_denseCopy
DenseScale	SUNDlsMat_DenseScale
denseScale	SUNDlsMat_denseScale
denseAddIdentity	SUNDlsMat_denseAddIdentity
DenseMatvec	SUNDlsMat_DenseMatvec
denseMatvec	SUNDlsMat_denseMatvec
BandGBTRF	SUNDlsMat_BandGBTRF
bandGBTRF	SUNDlsMat_bandGBTRF
BandGBTRS	SUNDlsMat_BandGBTRS
bandGBTRS	SUNDlsMat_bandGBTRS
BandCopy	SUNDlsMat_BandCopy
bandCopy	SUNDlsMat_bandCopy
BandScale	SUNDlsMat_BandScale
bandScale	SUNDlsMat_bandScale
bandAddIdentity	SUNDlsMat_bandAddIdentity
BandMatvec	SUNDlsMat_BandMatvec
bandMatvec	SUNDlsMat_bandMatvec
ModifiedGS	SUNModifiedGS
ClassicalGS	SUNClassicalGS
QRfact	SUNQRFact
QRsol	SUNQRsol
DlsMat_NewDenseMat	SUNDlsMat_NewDenseMat
DlsMat_NewBandMat	SUNDlsMat_NewBandMat
DestroyMat	SUNDlsMat_DestroyMat
NewIntArray	SUNDlsMat_NewIntArray
NewIndexArray	SUNDlsMat_NewIndexArray
NewRealArray	SUNDlsMat_NewRealArray
DestroyArray	SUNDlsMat_DestroyArray
AddIdentity	SUNDlsMat_AddIdentity
SetToZero	SUNDlsMat_SetToZero
PrintMat	SUNDlsMat_PrintMat
newDenseMat	SUNDlsMat_newDenseMat
newBandMat	SUNDlsMat_newBandMat
destroyMat	SUNDlsMat_destroyMat
newIntArray	SUNDlsMat_newIntArray
newIndexArray	SUNDlsMat_newIndexArray
newRealArray	SUNDlsMat_newRealArray
destroyArray	SUNDlsMat_destroyArray

In addition, the entire `sundials_lapack.h` header file is now deprecated for removal in SUNDIALS v7.0.0. Note, this header file is not needed to use the SUNDIALS LAPACK linear solvers.

Deprecated “bootstrap” and “minimum correction” predictors in `ARKStep` (options 4 and 5 to `ARKStepSetPredictorMethod()`) and the “bootstrap” predictor in `MRISStep` (option 4 to `MRISStepSetPredictorMethod()`). These functions will output a deprecation warning message and will be removed in a future release.

## 23.24 Changes to SUNDIALS in release 5.8.0

### New Features

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

New *SUNMatrix* and *SUNLinearSolver* implementation were added to interface with the Intel oneAPI Math Kernel Library (oneMKL). Both the matrix and the linear solver support general dense linear systems as well as block diagonal linear systems. See §10.9 for more details. This matrix is experimental and is subject to change from version to version.

Added a new *optional* function to the `SUNLinearSolver` API, `SUNLinSolSetZeroGuess()`, to indicate that the next call to `SUNLinSolSolve()` will be made with a zero initial guess. `SUNLinearSolver` implementations that do not use the `SUNLinSolNewEmpty()` constructor will, at a minimum, need set the `setzeroguess` function pointer in the linear solver ops structure to NULL. The SUNDIALS iterative linear solver implementations have been updated to leverage this new set function to remove one dot product per solve.

The time integrator packages (ARKODE, CVODE(S), and IDA(S)) all now support a new “matrix-embedded” *SUNLinearSolver* type. This type supports user-supplied `SUNLinearSolver` implementations that set up and solve the specified linear system at each linear solve call. Any matrix-related data structures are held internally to the linear solver itself, and are not provided by the SUNDIALS package.

Added functions to ARKODE and CVODE(S) for supplying an alternative right-hand side function and to IDA(S) for supplying an alternative residual for use within nonlinear system function evaluations:

- `ARKStepSetNlsRhsFn()`
- `MRISStepSetNlsRhsFn()`
- `CVodeSetNlsRhsFn()`
- `IDASetNlsResFn()`

Support for user-defined inner (fast) integrators has been to the `MRISStep` module. See *MRISStep Custom Inner Steppers* for more information on providing a user-defined integration method.

Added specialized fused HIP kernels to CVODE which may offer better performance on smaller problems when using CVODE with the HIP vector. See the optional input function `CVodeSetUseIntegratorFusedKernels()` for more information. As with other SUNDIALS HIP features, this capability is considered experimental and may change from version to version.

New KINSOL options have been added to apply a constant damping factor in the fixed point and Picard iterations (see `KINSetDamping()`), to delay the start of Anderson acceleration with the fixed point and Picard iterations (see `KINSetDelayAA()`), and to return the newest solution with the fixed point iteration (see `KINSetReturnNewest()`).

The installed `SUNDIALSConfig.cmake` file now supports the `COMPONENTS` option to `find_package`. The exported targets no longer have `IMPORTED_GLOBAL` set.

### Bug Fixes

A bug was fixed in `SUNMatCopyOps()` where the matrix-vector product setup function pointer was not copied.

A bug was fixed in the *SPBCGS* and *SPTFQMR* solvers for the case where a non-zero initial guess and a solution scaling vector are provided. This fix only impacts codes using SPBCGS or SPTFQMR as standalone solvers as all SUNDIALS packages utilize a zero initial guess.

A bug was fixed in the ARKODE stepper modules where the stop time may be passed after resetting the integrator.

A bug was fixed in `IDASSetJacTimesResFn()` in IDAS where the supplied function was used in the dense finite difference Jacobian computation rather than the finite difference Jacobian-vector product approximation.

A bug was fixed in the KINSOL Picard iteration where the value of `KINSetMaxSetupCalls()` would be ignored.

## 23.25 Changes to SUNDIALS in release 5.7.0

A new *N\_Vector* implementation based on the SYCL abstraction layer has been added targeting Intel GPUs. At present the only SYCL compiler supported is the DPC++ (Intel oneAPI) compiler. See §8.12 for more details. This vector is considered experimental and is subject to major changes even in minor releases.

A new *SUNMatrix* and *SUNLinearSolver* implementation were added to interface with the MAGMA linear algebra library. Both the matrix and the linear solver support general dense linear systems as well as block diagonal linear systems, and both are targeted at GPUs (AMD or NVIDIA). See §10.8 for more details.

## 23.26 Changes to SUNDIALS in release 5.6.1

Fixed a CMake bug which caused an error if the `CMAKE_CXX_STANDARD` and `SUNDIALS_RAJA_BACKENDS` options were not provided.

Fixed some compiler warnings when using the IBM XL compilers.

## 23.27 Changes to SUNDIALS in release 5.6.0

A new *N\_Vector* implementation based on the AMD ROCm HIP platform has been added. This vector can target NVIDIA or AMD GPUs. See §8.11 for more details. This vector is considered experimental and is subject to change from version to version.

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

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

The SUNDIALS matrix and linear solver interfaces to the *cuSparse matrix* and *cuSolver batched QR solver* no longer require using the CUDA *N\_Vector*. Instead, they require that the vector utilized provides the `N_VGetDeviceArrayPointer()` operation, and that the pointer returned by `N_VGetDeviceArrayPointer()` is a valid CUDA device pointer.

## 23.28 Changes to SUNDIALS in release 5.5.0

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

Added support for SuperLU DIST 6.3.0 or newer.

## 23.29 Changes to SUNDIALS in release 5.4.0

### Major Features

A new class, *SUNMemoryHelper*, was added to support **GPU users** who have complex memory management needs such as using memory pools. This is paired with new constructors for the CUDA and RAJA vectors that accept a *SUNMemoryHelper* object. Refer to §4.7, §16, §8.10 and §8.13 for more information.

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

An interface between ARKStep and the XBraid multigrid reduction in time (MGRIT) library [1] has been added to enable parallel-in-time integration. See the *Multigrid Reduction in Time with XBraid* section for more information and the example codes in `examples/arkode/CXX_xbraid`. This interface required the addition of three new *N\_Vector* operations to exchange vector data between computational nodes, see *N\_VBufSize()*, *N\_VBufPack()*, and *N\_VBufUnpack()*. These *N\_Vector* operations are only used within the XBraid interface and need not be implemented for any other context.

### New Features

The *RAJA vector* has been updated to mirror the CUDA vector. Notably, the update adds managed memory support to the RAJA vector. Users of the vector will need to update any calls to the *N\_VMake\_Raja()* function because that signature was changed. This vector remains experimental and is subject to change from version to version.

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

- *ARKStepGetNumNonlinSolvIters()*
- *ARKStepGetNumNonlinSolvConvFails()*
- *ARKStepGetNonlinSolvStats()*
- *MRISStepGetNumNonlinSolvIters()*
- *MRISStepGetNumNonlinSolvConvFails()*
- *MRISStepGetNonlinSolvStats()*
- *CVodeGetNumNonlinSolvIters()*
- *CVodeGetNumNonlinSolvConvFails()*
- *CVodeGetNonlinSolvStats()*
- *IDAGetNumNonlinSolvIters()*
- *IDAGetNumNonlinSolvConvFails()*
- *IDAGetNonlinSolvStats()*

Added the following the following functions that advanced users might find useful when providing a custom *SUNNonlinSolSysFn()*:

- *ARKStepComputeState()*
- *ARKStepGetNonlinearSystemData()*

- `MRIStepComputeState()`
- `MRIStepGetNonlinearSystemData()`
- `CVodeComputeState()`
- `CVodeGetNonlinearSystemData()`
- `IDAGetNonlinearSystemData()`

Added new functions to CVODE(S), ARKODE, and IDA(S) to specify the factor for converting between integrator tolerances (WRMS norm) and linear solver tolerances (L2 norm) i.e.,  $\text{tol\_L2} = \text{nrmfac} * \text{tol\_WRMS}$ :

- `ARKStepSetLSNormFactor()`
- `ARKStepSetMassLSNormFactor()`
- `MRISetLSNormFactor()`
- `CVodeSetLSNormFactor()`
- `IDASetLSNormFactor()`

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

Updated the MRISet time-stepping module in ARKODE to support higher-order MRI-GARK methods [93], including methods that involve solve-decoupled, diagonally-implicit treatment of the slow time scale.

The function `CVodeSetLSetupFrequency()` has been added to CVODE(S) to set the frequency of calls to the linear solver setup function.

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

Added support for CUDA 11.

### Bug Fixes

A minor inconsistency in CVODE(S) and a bug ARKODE when checking the Jacobian evaluation frequency has been fixed. As a result codes using a non-default Jacobian update frequency through a call to `CVodeSetMaxStepsBetweenJac` or `ARKStepSetMaxStepsBetweenJac` will need to increase the provided value by 1 to achieve the same behavior as before.

In IDAS and CVODES, the functions for forward integration with checkpointing (`IDASolveF()`, `CVodeF()`) are now subject to a restriction on the number of time steps allowed to reach the output time. This is the same restriction applied to `IDASolve()` and `CVode()`. The default maximum number of steps is 500, but this may be changed using the `CVodeSetMaxNumSteps()` and `IDASetMaxNumSteps()` function. This change fixes a bug that could cause an infinite loop in `IDASolveF()` and `CVodeF()`. **This change may cause a runtime error in existing user code.**

Fixed bug in using ERK method integration with static mass matrices.

### Deprecation Notice

For greater clarity the following functions have been deprecated:

- `CVodeSetMaxStepsBetweenJac`
- `ARKStepSetMaxStepsBetweenJac`
- `ARKStepSetMaxStepsBetweenLSet`

The following functions should be used instead:

- `CVodeSetJacEvalFrequency()`

- [`ARKStepSetJacEvalFrequency\(\)`](#)
- [`ARKStepSetLSetupFrequency\(\)`](#)

## 23.30 Changes to SUNDIALS in release 5.3.0

### Major Feature

Added support to CVODE for integrating IVPs with constraints using BDF methods and projecting the solution onto the constraint manifold with a user defined projection function. This implementation is accompanied by additions to user documentation and CVODE examples. See [`CVodeSetProjFn\(\)`](#) for more information.

### New Features

Added the ability to control the CUDA kernel launch parameters for the CUDA vector and sparse matrix implementations. These implementations remain experimental and are subject to change from version to version. In addition, the CUDA vector kernels were rewritten to be more flexible. Most users should see equivalent performance or some improvement, but a select few may observe minor performance degradation with the default settings. Users are encouraged to contact the SUNDIALS team about any performance changes that they notice.

Added new capabilities for monitoring the solve phase in the Newton and fixed-point [`SUNNonlinearSolver`](#), and the SUNDIALS iterative linear solvers. SUNDIALS must be built with the CMake option `SUNDIALS_BUILD_WITH_MONITORING` to use these capabilities.

Added specialized fused CUDA kernels to CVODE which may offer better performance on smaller problems when using CVODE with the CUDA vector. See the optional input function [`CVodeSetUseIntegratorFusedKernels\(\)`](#) for more information. As with other SUNDIALS CUDA features, this is feature is experimental and may change from version to version.

Added a new function, [`CVodeSetMonitorFn\(\)`](#), that takes a user-function to be called by CVODE after every `nst` successfully completed time-steps. This is intended to provide a way of monitoring the CVODE statistics throughout the simulation.

Added a new function [`CVodeGetLinSolveStats\(\)`](#) to get the CVODE linear solver statistics as a group.

Added the following optional functions to provide an alternative ODE right-hand side function (ARKODE and CVODE(S)), DAE residual function (IDA(S)), or nonlinear system function (KINSOL) for use when computing Jacobian-vector products with the internal difference quotient approximation:

- [`ARKStepSetJacTimesRhsFn\(\)`](#)
- [`CVodeSetJacTimesRhsFn\(\)`](#)
- [`CVodeSetJacTimesRhsFnB\(\)`](#)
- [`IDASetJacTimesResFn\(\)`](#)
- [`IDASetJacTimesResFnB\(\)`](#)
- [`KINSetJacTimesVecSysFn\(\)`](#)

### Bug Fixes

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

Fixed a bug in ARKODE where the prototypes for [`ERKStepSetMinReduction\(\)`](#) and [`ARKStepSetMinReduction\(\)`](#) were not included in `arkode_erkstep.h` and `arkode_arkstep.h` respectively.

Fixed a bug in ARKODE where inequality constraint checking would need to be disabled and then re-enabled to update the inequality constraint values after resizing a problem. Resizing a problem will now disable constraints and a call



to [ARKStepSetConstraints\(\)](#) or [ERKStepSetConstraints\(\)](#) is required to re-enable constraint checking for the new problem size.

## 23.31 Changes to SUNDIALS in release 5.2.0

### New Features

The following functions were added to each of the time integration packages to enable or disable the scaling applied to linear system solutions with matrix-based linear solvers to account for lagged matrix information:

- [ARKStepSetLinearSolutionScaling\(\)](#)
- [CNodeSetLinearSolutionScaling\(\)](#)
- [CNodeSetLinearSolutionScalingB\(\)](#)
- [IDASSetLinearSolutionScaling\(\)](#)
- [IDASSetLinearSolutionScalingB\(\)](#)

When using a matrix-based linear solver with ARKODE, IDA(S), or BDF methods in CVODE(S) scaling is enabled by default.

Added a new [SUNMatrix](#) implementation that interfaces to the sparse matrix implementation from the NVIDIA cuSPARSE library, see §9.7 for more details. In addition, the CUDA Sparse linear solver has been updated to use the new matrix, as such, users of this matrix will need to update their code. This implementations are still considered to be experimental, thus they are subject to breaking changes even in minor releases.

Added a new “stiff” interpolation module to ARKODE, based on Lagrange polynomial interpolation, that is accessible to each of the ARKStep, ERKStep and MRISStep time-stepping modules. This module is designed to provide increased interpolation accuracy when integrating stiff problems, as opposed to the ARKODE-standard Hermite interpolation module that can suffer when the IVP right-hand side has large Lipschitz constant. While the Hermite module remains the default, the new Lagrange module may be enabled using one of the routines [ARKStepSetInterpolantType\(\)](#), [ERKStepSetInterpolantType\(\)](#), or [MRISStepSetInterpolantType\(\)](#). The serial example problem `ark_brusselator.c` has been converted to use this Lagrange interpolation module. Created accompanying routines [ARKStepSetInterpolantDegree\(\)](#), [ERKStepSetInterpolantDegree\(\)](#) and [MRISStepSetInterpolantDegree\(\)](#) to provide user control over these interpolating polynomials.

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

### Bug Fixes

Fixed a build system bug related to the Fortran 2003 interfaces when using the IBM XL compiler. When building the Fortran 2003 interfaces with an XL compiler it is recommended to set [CMAKE\\_Fortran\\_COMPILER](#) to `f2003`, `xl_f2003`, or `xl_f2003_r`.

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

Fixed a linkage bug affecting Windows users that stemmed from `dllimport/dllexport` attribute missing on some SUNDIALS API functions.

Fixed a memory leak in CVODES and IDAS from not deallocating the `atolSmin0` and `atolQsmin0` arrays.

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

### Deprecation Notice



The routines [ARKStepSetDenseOrder\(\)](#), [ARKStepSetDenseOrder\(\)](#) and [ARKStepSetDenseOrder\(\)](#) have been deprecated and will be removed in a future release. The new functions [ARKStepSetInterpolantDegree\(\)](#), [StepSetInterpolantDegree\(\)](#), and [ARKStepSetInterpolantDegree\(\)](#) should be used instead.

## 23.32 Changes to SUNDIALS in release 5.1.0

### New Features

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

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

Added support for damping when using Anderson acceleration in KINSOL. See the [Mathematical Considerations](#) and the description of the [KINSetDampingAA\(\)](#) function for more details.

Added support for constant damping to the fixed-point [SUNNonlinearSolver](#) when using Anderson acceleration. See [SUNNonlinSol\\_FixedPoint description](#) and the [SUNNonlinSolSetDamping\\_FixedPoint\(\)](#) for more details.

Added two utility functions, [SUNDIALSFileOpen\(\)](#) and [SUNDIALSFileClose\(\)](#) for creating/destroying file pointers. These are useful when using the Fortran 2003 interfaces.

Added a new build system option, `CUDA_ARCH`, to specify the CUDA architecture to target.

### Bug Fixes

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

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

Fixed a build system bug related to finding PETSc when using the CMake variables [PETSC\\_INCLUDES](#) and [PETSC\\_LIBRARIES](#) instead of [PETSC\\_DIR](#).

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

## 23.33 Changes to SUNDIALS in release 5.0.0

### Build System

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

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

### NVector

Two new functions were added to aid in creating custom [N\\_Vector](#) objects. The constructor [N\\_VNewEmpty\(\)](#) allocates an “empty” generic [N\\_Vector](#) with the object’s content pointer and the function pointers in the operations structure initialized to NULL. When used in the constructor for custom objects this function will ease the introduction of any

new optional operations to the *N\_Vector* API by ensuring only required operations need to be set. Additionally, the function *N\_VCopyOps()* has been added to copy the operation function pointers between vector objects. When used in clone routines for custom vector objects these functions also will ease the introduction of any new optional operations to the *N\_Vector* API by ensuring all operations are copied when cloning objects.

Added new *N\_Vector* implementations, *ManyVector* and *MPIManyVector*, to support flexible partitioning of solution data among different processing elements (e.g., CPU + GPU) or for multi-physics problems that couple distinct MPI-based simulations together (see the §8.17 and §8.18 for more details). This implementation is accompanied by additions to user documentation and SUNDIALS examples.

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

One new required vector operation and ten new optional vector operations have been added to the *N\_Vector* API. The new required operation, *N\_VGetLength()*, returns the global vector length. The optional operations have been added to support the new *MPIManyVector* implementation. The operation *N\_VGetCommunicator()* must be implemented by subvectors that are combined to create an *MPIManyVector*, but is not used outside of this context. The remaining nine operations are optional local reduction operations intended to eliminate unnecessary latency when performing vector reduction operations (norms, etc.) on distributed memory systems. The optional local reduction vector operations are *N\_VDotProdLocal*, *N\_VMaxNormLocal*, *N\_VMinLocal*, *N\_VL1NormLocal*, *N\_VWSqrSumLocal*, *N\_VWSqrSumMaskLocal*, *N\_VInvTestLocal*, *N\_VConstrMaskLocal*, and *N\_VMinQuotientLocal*. If an *N\_Vector* implementation defines any of the local operations as NULL, then the *MPIManyVector* will call standard *N\_Vector* operations to complete the computation.

The \*\_MPICuda and \*\_MPIRaja functions have been removed from the CUDA and RAJA vector implementations respectively. Accordingly, the *nvector\_mpicuda.h*, *nvector\_mpiraja.h*, *libsundials\_nvecmpicuda.lib*, and *libsundials\_nvecmpicudaraja.lib* files have been removed. Users should use the MPI+X vector in conjunction with the CUDA and RAJA vectors to replace the functionality. The necessary changes are minimal and should require few code modifications. See the example programs in *examples/ida/mpicuda* and *examples/ida/mpiraja* for examples of how to use the MPI+X vector with the CUDA and RAJA vectors, respectively.

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

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

Added three new *N\_Vector* utility functions, *N\_VGetVecAtIndexVectorArray()*, *N\_VSetVecAtIndexVectorArray()*, and *N\_VNewVectorArray()*, for working with *N\_Vector* arrays when using the Fortran 2003 interfaces.

## SUNMatrix

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

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

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

A new *SUNMatrix* (and *SUNLinearSolver*) implementation was added to facilitate the use of the SuperLU\_DIST library with SUNDIALS.

### SUNLinearSolver

A new function was added to aid in creating custom *SUNLinearSolver* objects. The constructor *SUNLinSol-NewEmpty()* allocates an “empty” generic *SUNLinearSolver* with the object’s content pointer and the function pointers in the operations structure initialized to NULL. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the *SUNLinearSolver* API by ensuring only required operations need to be set.

The return type of the *SUNLinSolLastFlag* in the *SUNLinearSolver* has changed from long int to *sunindextype* to be consistent with the type used to store row indices in dense and banded linear solver modules.

Added a new optional operation to the *SUNLinearSolver* API, *SUNLinSolGetID()*, that returns a *SUNLinearSolver\_ID* for identifying the linear solver module.

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

A new *SUNLinearSolver* (and *SUNMatrix*) implementation was added to facilitate the use of the SuperLU\_DIST library with SUNDIALS.

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

Added three new accessor functions to the KLU linear solver to provide user access to the underlying KLU solver structures: *SUNLinSol\_KLUGetSymbolic()*, *SUNLinSol\_KLUGetNumeric()*, and *SUNLinSol\_KLUGetCommon()*.

### SUNNonlinearSolver

A new function was added to aid in creating custom *SUNNonlinearSolver* objects. The constructor *SUNNonlinSol-NewEmpty()* allocates an “empty” generic *SUNNonlinearSolver* with the object’s content pointer and the function pointers in the operations structure initialized to NULL. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the *SUNNonlinearSolver* API by ensuring only required operations need to be set.

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

The inputs values passed to the first two inputs of the *SUNNonlinSolSolve()* function in the *SUNNonlinearSolver* have been changed to be the predicted state and the initial guess for the correction to that state. Additionally, the definitions of *SUNNonlinSolSetupFn()* and *SUNNonlinSolSolveFn()* in the *SUNNonlinearSolver* API have been updated to remove unused input parameters. For more information on the nonlinear system formulation and the API functions see *Nonlinear Algebraic Solvers*.

Added a new *SUNNonlinearSolver* implementation for interfacing with the *PETSc SNES* nonlinear solver.

### New Features

A new linear solver interface functions, *ARKLsLinSysFn* and *CVLsLinSysFn*, as added as an alternative method for evaluating the linear systems  $M - \gamma J$  or  $I - \gamma J$ .

Added the following functions to get the current state and gamma value to ARKStep, CVODE and CVODES that may be useful to users who choose to provide their own nonlinear solver implementation:

- *ARKStepGetCurrentState()*
- *ARKStepGetCurrentGamma()*
- *CVodeGetCurrentGamma()*
- *CVodeGetCurrentState()*

- `CNodeGetCurrentGamma()`
- `CNodeGetCurrentStateSens()`
- `CNodeGetCurrentSensSolveIndex()`
- `IDAGetCurrentCj()`
- `IDAGetCurrentY()`
- `IDAGetCurrentYp()`
- `IDAComputeY()`
- `IDAComputeYp()`

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

The ARKODE, CVODE(S), IDA(S), and KINSOL linear solver interfaces have been updated to only zero the Jacobian matrix before calling a user-supplied Jacobian evaluation function when the attached linear solver has type `SUNLIN-EARSOLVER_DIRECT`.

Added new Fortran 2003 interfaces to all of the SUNDIALS packages (ARKODE, CVODE(S), IDA(S), and KINSOL) as well as most of the `N_Vector`, `SUNMatrix`, `SUNLinearSolver`, and `SUNNonlinearSolver` implementations. See §20 section for more details. These new interfaces were generated with SWIG-Fortran and provide a user an idiomatic Fortran 2003 interface to most of the SUNDIALS C API.

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

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

Added two new embedded ARK methods of orders 4 and 5 to ARKODE (from [71]).

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

Add two new ‘Set’ functions to MRISStep, `MRISStepSetPreInnerFn()` and `MRISStepSetPostInnerFn()`, for performing communication or memory transfers needed before or after the inner integration.

### Bug Fixes

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

Fixed a memory leak in the PETSc `N_Vector` clone function.

Fixed a memory leak in the ARKODE, CVODE, and IDA F77 interfaces when not using the default nonlinear solver.

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

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

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

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

Fixed a bug in the CVODE and CVODES constraint handling where the step size could be set below the minimum step size.

Fixed a bug in the CVODE and CVODES nonlinear solver interfaces where the norm of the accumulated correction was not updated when using a non-default convergence test function.

Fixed a bug in the CVODES `cvRescale` function where the loops to compute the array of scalars for the fused vector scale operation stopped one iteration early.

Fixed a bug in CVODES and IDAS where `CVodeF()` and `IDASolveF()` would return the wrong flag under certain circumstances.

Fixed a bug in CVODES and IDAS where `CVodeF()` and `IDASolveF()` would not return a root in `NORMAL_STEP` mode if the root occurred after the desired output time.

Fixed a bug in the IDA and IDAS linear solver interfaces where an incorrect Jacobian-vector product increment was used with iterative solvers other than SPGMR and SPFGMR.

Fixed a bug the IDAS `IDAQuadReInitB()` function where an incorrect memory structure was passed to `IDAQuadReInit()`.

Fixed a bug in the KINSOL linear solver interface where the auxiliary scalar `sJpnorm` was not computed when necessary with the Picard iteration and the auxiliary scalar `sFdotJp` was unnecessarily computed in some cases.

## 23.34 Changes to SUNDIALS in release 4.1.0

### Removed Implementation Headers

The implementation header files (`*_impl.h`) are no longer installed. This means users who are directly accessing or manipulating package memory structures will need to update their code to use the package's public API.

### New Features

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

### Bug Fixes

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

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

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

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

## 23.35 Changes to SUNDIALS in release 4.0.2

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

Moved the definitions of backwards compatibility functions for the prior direct linear solver (DLS) and scaled preconditioned iterative linear solvers (SPILS) to a source file. The symbols are now included in the appropriate package library, e.g. `libsundials_cvode.lib`.

## 23.36 Changes to SUNDIALS in release 4.0.1

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

## 23.37 Changes to SUNDIALS in release 4.0.0

The direct and iterative linear solver interfaces in all SUNDIALS packages have been merged into a single unified linear solver interface to support any valid *SUNLinearSolver*. This includes the `DIRECT` and `ITERATIVE` types as well as the new `MATRIX_ITERATIVE` type. Details regarding how SUNDIALS packages utilize linear solvers of each type as well as a discussion regarding the intended use cases for user-supplied linear solver implementations are included in §10. All example programs have been updated to use the new unified linear solver interfaces.

The unified linear solver interface is very similar to the previous DLS (direct linear solver) and SPILS (scaled pre-conditioned iterative linear solver) interface in each package. To minimize challenges in user migration to the unified linear solver interfaces, the previous DLS and SPILS functions may still be used however, these are now deprecated and will be removed in a future release. Additionally, that Fortran users will need to enlarge their array of optional integer outputs, and update the indices that they query for certain linear solver related statistics.

The names of all SUNDIALS-provided *SUNLinearSolver* constructors have been updated to follow the naming convention `SUNLinSol_*` where `*` is the name of the linear solver. The new constructor names are:

- *SUNLinSol\_Band()*
- *SUNLinSol\_Dense()*
- *SUNLinSol\_KLU()*
- *SUNLinSol\_LapackBand()*
- *SUNLinSol\_LapackDense()*
- *SUNLinSol\_PCG()*
- *SUNLinSol\_SPBCGS()*
- *SUNLinSol\_SPFGMV()*
- *SUNLinSol\_SPGMR()*
- *SUNLinSol\_SPTFQMR()*
- *SUNLinSol\_SuperLUMT()*

Linear solver-specific “set” routine names have been similarly standardized. To minimize challenges in user migration to the new names, the previous function names may still be used however, these are now deprecated and will be removed in a future release. All example programs and the standalone linear solver examples have been updated to use the new naming convention.

The *SUNLinSol\_Band()* constructor has been simplified to remove the storage upper bandwidth argument.

SUNDIALS integrators (ARKODE, CVODE(S), and IDA(S)) have been updated to utilize generic nonlinear solvers defined by the *SUNNonlinearSolver* API. This enables the addition of new nonlinear solver options and allows for external or user-supplied nonlinear solvers. The nonlinear solver API and SUNDIALS provided implementations are described in *Nonlinear Algebraic Solvers* and follow the same object oriented design used by the *N\_Vector*, *SUNMatrix*, and *SUNLinearSolver* classes. Currently two nonlinear solver implementations are provided, *Newton* and *fixed-point*. These replicate the previous integrator-specific implementations of Newton’s method and a fixed-point iteration (previously referred to as a functional iteration), respectively. Note the new *fixed-point* implementation can optionally utilize Anderson’s method to accelerate convergence. Example programs using each of these nonlinear solvers in a standalone manner have been added and all example programs have been updated accordingly.



The SUNDIALS integrators (ARKODE, CVODE(S), and IDA(S)) all now use the *Newton SUNNonlinearSolver* by default. Users that wish to use the *fixed-point SUNNonlinearSolver* will need to create the corresponding nonlinear solver object and attach it to the integrator with the appropriate set function:

- `ARKStepSetNonlinearSolver()`
- `CVodeSetNonlinearSolver()`
- `IDASSetNonlinearSolver()`

Functions for setting the nonlinear solver options or getting nonlinear solver statistics remain unchanged and internally call generic `SUNNonlinearSolver` functions as needed.

With the introduction of the *SUNNonlinearSolver* class, the input parameter `iter` to `CVodeCreate()` has been removed along with the function `CVodeSetIterType` and the constants `CV_NEWTON` and `CV_FUNCTIONAL`. While SUNDIALS includes a fixed-point nonlinear solver, it is not currently supported in IDA.

Three fused vector operations and seven vector array operations have been added to the *N\_Vector* API. These *optional* operations are disabled by default and may be activated by calling vector specific routines after creating a vector (see §8.1 for more details). The new operations are intended to increase data reuse in vector operations, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. The fused operations are:

- `N_VLinearCombination()`
- `N_VScaleAddMulti()`
- `N_VDotProdMulti()`

and the vector array operations are:

- `N_VLinearCombinationVectorArray()`
- `N_VScaleVectorArray()`
- `N_VConstVectorArray()`
- `N_VWrmsNormVectorArray()`
- `N_VWrmsNormMaskVectorArray()`
- `N_VScaleAddMultiVectorArray()`
- `N_VLinearCombinationVectorArray()`

If an *N\_Vector* implementation defines the implementation any of these operations as `NULL`, then standard vector operations will automatically be called as necessary to complete the computation.

A new *N\_Vector* implementation, *OpenMPDEV*, leveraging OpenMP device offloading has been added.

Multiple updates to the *CUDA* vector were made:

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

- Added `N_VNewManaged_Cuda()`, `N_VMakeManaged_Cuda()`, and `N_VIsManagedMemory_Cuda()` functions to accommodate using managed memory with the CUDA vector.

Multiple updates to the `RAJA` vector were made:

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

Two changes were made in the ARKODE and CVODE(S) initial step size algorithm:

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

Fortran 2003 interfaces to CVODE, the fixed-point and Newton nonlinear solvers, the dense, band, KLU, PCG, SP-BCGS, SPFGMR, SPGMR, and SPTFQMR linear solvers, and the serial, PThreads, and OpenMP vectors have been added.

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

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

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

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

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

## 23.38 Changes to SUNDIALS in release 3.2.1

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

Fixed the library installation path for multiarch systems. This fix changes the default library installation path from `CMAKE_INSTALL_PREFIX/lib` to `CMAKE_INSTALL_PREFIX/CMAKE_INSTALL_LIBDIR`. The default value library directory name is automatically set to `lib`, `lib64`, or `lib/<multiarch-tuple>` depending on the system, but maybe be overridden by setting `CMAKE_INSTALL_LIBDIR`.



## 23.39 Changes to SUNDIALS in release 3.2.0

### Library Name Change

Changed the name of the RAJA nvector library from `libsundials_nvecraja.lib` to `libsundials_nveccudaraja.lib` to better reflect that the RAJA vector only support the CUDA backend currently.

### New Features

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.

Support for optional inequality constraints on individual components of the solution vector has been added to CVODE and CVODES. For more details see the [Mathematical Considerations](#) and [Optional input functions](#) sections. Use of `CVodeSetConstraints()` requires the *N\_Vector* operations `N_VMinQuotient()`, `N_VConstrMask()`, and `N_VCompare()` that were not previously required by CVODE and CVODES.

### CMake Updates

CMake 3.1.3 is now the minimum required CMake version.

Deprecated 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 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., `ENABLE_LAPACK` is ON) the build system will infer the scheme from the Fortran compiler. If a Fortran compiler is not available or the inferred or default scheme needs to be overridden, the advanced options `SUNDIALS_F77_FUNC_CASE` and `SUNDIALS_F77_FUNC_UNDERSCORES` can be used to manually set the name-mangling scheme and bypass trying to infer the scheme.

Parts of the main `CMakeLists.txt` file were moved to new files in the `src` and `example` directories to make the CMake configuration file structure more modular.

### Bug Fixes

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

Fixed a thread-safety issue in CVODES and IDAS when using adjoint sensitivity analysis.

Fixed a bug in IDAS where the saved residual value used in the nonlinear solve for consistent initial conditions was passed as temporary workspace and could be overwritten.

## 23.40 Changes to SUNDIALS in release 3.1.2

### CMake Updates

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

### New Features

Added the function `SUNSparseMatrix_Reallocate()` to allow specification of the matrix nonzero storage.

Added named constants for the two reinitialization types for the KLU SUNLinearSolver.

Updated the `SUNMatScaleAdd()` and `SUNMatScaleAddI()` implementations in the sparse SUNMatrix 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  $M + \gamma J$  or  $M + \gamma J$  manually (with zero entries if needed).

The following examples from the usage notes page of the SUNDIALS website, and updated them to work with SUNDIALS 3.x:

- `cvDisc_dns.c` demonstrates using CVODE with discontinuous solutions or RHS.
- `cvRoberts_dns_negsol.c` illustrates the use of the RHS function return value to control unphysical negative concentrations.
- `cvRoberts_FSA_dns_Switch.c` demonstrates switching on/off forward sensitivity computations. This example came from the usage notes page of the SUNDIALS website.

### Bug Fixes

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

Fixed a bug in the full KLU SUNLinearSolver reinitialization approach where the sparse SUNMatrix pointer would go out of scope on some architectures.

The misnamed function `CVSpilsSetJacTimesSetupFnBS` has been deprecated and replaced by `CVSpilsSetJacTimesBS`. The deprecated function `CVSpilsSetJacTimesSetupFnBS` will be removed in the next major release.

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

## 23.41 Changes to SUNDIALS in release 3.1.1

### Bug Fixes

Fixed a minor bug in the CVODE and CVODES `cvSLdet`, where a return was missing in the error check for three inconsistent roots.

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 `ARKReInit`, 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 the KLU SUNLinearSolver to use a typedef for the precision-specific solve functions to avoid compiler warnings.

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

Fixed bug in the sparse SUNMatrix where `int` was used instead of `sunindextype` in one location.

Fixed a minor bug in `KINPrintInfo` where a case was missing for `KIN_REPTD_SYSFUNC_ERR` leading to an undefined info message.

Added missing `#include <stdio.h>` in `N_Vector` and `SUNMatrix` header files.

Added missing prototypes for `ARKSpilsGetNumMTSetups` in ARKODE and `IDASpilsGetNumJTSetupEvals` in IDA and IDAS.

Fixed an indexing bug in the CUDA vector implementation of `N_VWrmsNormMask()` and revised the RAJA vector 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` e.g., iterative linear solvers, explicit methods in ARKODE, functional iteration in CVODE, etc.

## 23.42 Changes to SUNDIALS in release 3.1.0

Added `N_Vector` 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.

## 23.43 Changes to SUNDIALS in release 3.0.0

### Major Feature

Added new linear solver and matrix interfaces for all SUNDIALS packages and updated the existing linear solver and matrix implementations. The goal of the redesign is to provide greater encapsulation and ease interfacing custom linear solvers with linear solver libraries. Specific changes include:

- Added a `SUNMatrix` interface with three provided implementations: dense, banded, and sparse. These replicate previous SUNDIALS direct (DIs) and sparse (SIs) matrix structures.
- Added example problems demonstrating use of the matrices.
- Added a `SUNLinearSolver` interface 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.
- Added example problems demonstrating use of the linear solvers.
- Expanded package-provided direct linear solver (DIs) interfaces and scaled, preconditioned, iterative linear solver (SpIs) interfaces to utilize `SUNMatrix` and `SUNLinearSolver` objects.
- Removed package-specific, linear solver-specific, solver modules (e.g., CVDENSE, KINBAND, IDAKLU, ARK-SPGMR) since their functionality is entirely replicated by the generic DIs/SpIs interfaces and `SUNLinearSolver` / `SUNMatrix` classes. The exception is CVDIAG, a diagonal approximate Jacobian solver available to CVODE and CVODES.
- Converted all SUNDIALS example problems to utilize new the new matrix and linear solver objects, along with updated DIs and SpIs linear solver interfaces.
- Added SpIs 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.

Corresponding updates were made to all the example programs.

### New Features

`CUDA` and `RAJA N_Vector` implementations to support GPU systems. 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. For further information about RAJA, users are referred to the [RAJA web site](#).

Added the type *sunindextype* to support using 32-bit or 64-bit integer types for indexing arrays within all SUNDIALS structures. *sunindextype* is defined to `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 the sparse matrix interface that now uses *sunindextype*. Interfaces to PETSc, hypre, SuperLU\_MT, and KLU have been updated with 32-bit or 64-bit capabilities depending how the user configures SUNDIALS.

To avoid potential namespace conflicts, the macros defining `boolean_t` 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 support for many xSDK-compliant build system keys. For more information on xSDK compliance the *xSDK policies*. 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. For more information visit the *xSDK web site*.

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

Added comments to `arkode_butcher.c` regarding which methods should have coefficients accurate enough for use in quad precision.

### Build System

Renamed CMake options to enable/disable examples for greater clarity and added option to enable/disable Fortran 77 examples:

- Changed `EXAMPLES_ENABLE` to `EXAMPLES_ENABLE_C`
- Changed `CXX_ENABLE` to `EXAMPLES_ENABLE_CXX`
- Changed `F90_ENABLE` to `EXAMPLES_ENABLE_F90`
- Added `EXAMPLES_ENABLE_F77` option

Added separate `BLAS_ENABLE` and `BLAS_LIBRARIES` CMake variables.

Fixed minor CMake bugs and included additional error checking during CMake configuration.

### Bug Fixes

#### ARKODE

Fixed `RCONST` usage in `arkode_butcher.c`.

Fixed bug in `arkInitialSetup` to ensure the mass matrix vector product is set up before the “msetup” routine is called.

Fixed ARKODE `printf`-related compiler warnings when building SUNDIALS with extended precision.

#### CVODE and CVODES

*CVodeFree()* now calls `lfree` unconditionally (if non-NULL).

#### IDA and IDAS

Added missing prototype for *IDASetMaxBacksIC()* in `ida.h` and `idas.h`.

#### KINSOL

Corrected KINSOL Fortran name translation for `FKIN_SPFGMR`.

Renamed `KINLocalFn` and `KINCommFn` to *KINBBDDLocalFn* and *KINBBDDCommFn* respectively in the BBD preconditioner module for consistency with other SUNDIALS solvers.

## 23.44 Changes to SUNDIALS in release 2.7.0

### New Features and Enhancements

Two additional *N\_Vector* implementations were added – one for *hybre parallel vectors* and one for *PETSc vectors*. These additions are accompanied by additions to various interface functions and to user documentation.

Added a new *N\_Vector* function, *N\_VGetVectorID()*, that returns an identifier for the vector.

The sparse matrix structure was enhanced to support both CSR and CSC matrix storage formats.

Various additions were made to the KLU and SuperLU\_MT sparse linear solver interfaces, including support for the CSR matrix format when using KLU.

In all packages, the linear solver and preconditioner *free* routines were updated to return an integer.

In all packages, example codes were updated to use *N\_VGetArrayPointer\_\** rather than the *NV\_DATA* macro when using the native vectors shipped with SUNDIALS.

Additional example programs were added throughout including new examples utilizing the OpenMP vector.

### ARKODE

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 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 with different stage times 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.

### IDA and IDAS

The optional input function *IDASetMaxBacksIC()* was added to set the maximum number of linesearch backtracks in the initial condition calculation.

### Bug Fixes

Various minor fixes to installation-related files.

Fixed some examples with respect to the change to use new macro/function names e.g., *SUNRexp*, etc.

In all packages, a memory leak was fixed in the banded preconditioner and banded-block-diagonal preconditioner interfaces.

Corrected name *N\_VCloneEmptyVectorArray* to *N\_VCloneVectorArrayEmpty* in all documentation files.

Various corrections were made to the interfaces to the sparse solvers KLU and SuperLU\_MT.

For each linear solver, the various solver performance counters are now initialized to 0 in both the solver specification function and in the solver *linit* function. This ensures that these solver counters are initialized upon linear solver instantiation as well as at the beginning of the problem solution.

### ARKODE

The missing *ARKSpilsGetNumMtimesEvals* function was added – this had been included in the previous documentation but had not been implemented.

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.

#### *CVODE*

Corrections were made to three Fortran interface functions.

In FCVODE, fixed argument order bugs in the FCVKLU and FCVSUPERLUMT linear solver interfaces.

Added missing Fortran interface routines for supplying a sparse Jacobian routine with sparse direct solvers.

#### *CVODES*

A bug was fixed in the interpolation functions used in solving backward problems for adjoint sensitivity analysis.

In the interpolation routines for backward problems, added logic to bypass sensitivity interpolation if input sensitivity argument is NULL.

Changed each the return type of \*FreeB functions to `int` and added `return(0)` to each.

#### *IDA*

Corrections were made to three Fortran interface functions.

Corrected the output from the `idaFoodWeb_bnd.c` example, the wrong component was printed in `PrintOutput`.

#### *IDAS*

In the interpolation routines for backward problems, added logic to bypass sensitivity interpolation if input sensitivity argument is NULL.

Changed each the return type of \*FreeB functions to `int` and added `return(0)` to each.

Corrections were made to three Fortran interface functions.

Added missing Fortran interface routines for supplying a sparse Jacobian routine with sparse direct solvers.

#### *KINSOL*

The Picard iteration return was changed to always return the newest iterate upon success.

A minor bug in the line search was fixed to prevent an infinite loop when the beta condition fails and lambda is below the minimum size.

Corrections were made to three Fortran interface functions.

The functions `FKINCREATE` and `FKININIT` were added to split the `FKINMALLOC` routine into two pieces. `FKINMALLOC` remains for backward compatibility, but documentation for it has been removed.

Added missing Fortran interface routines for supplying a sparse Jacobian routine with sparse direct solvers.

#### **Matlab Interfaces Removed**

Removed the Matlab interface from distribution as it has not been updated since 2009.

## 23.45 Changes to SUNDIALS in release 2.6.2

### New Features and Enhancements

Various minor fixes to installation-related files

In KINSOL and ARKODE, updated the Anderson acceleration implementation with QR updating.

In CVODES and IDAS, added `ReInit` and `SetOrdering` wrappers for backward problems.

In IDAS, fixed for-loop bugs in `IDAackpntAllocVectors` that could lead to a memory leak.

### Bug Fixes

Updated the BiCGStab linear solver to remove a redundant dot product call.

Fixed potential memory leak in KLU `ReInit` functions in all solvers.

In ARKODE, fixed a bug in the Cash-Karp Butcher table where the method and embedding coefficient were swapped.

In ARKODE, fixed error in `arkDoErrorTest` in recovery after failure.

In CVODES, added CVKLUB prototype and corrected CVSuperLUMTB prototype.

In the CVODES and IDAS header files, corrected documentation of backward integration functions, especially the `which` argument.

In IDAS, added missing backward problem support functions `IDALapackDenseB`, `IDALapackDenseFreeB`, `IDALapackBandB`, and `IDALapackBandFreeB`.

In IDAS, made SuperLUMT call for backward problem consistent with CVODES.

In CVODE, IDA, and ARKODE, fixed Fortran interfaces to enable calls to `GetErrWeights`, `GetEstLocalErrors`, and `GetDky` within a time step.

## 23.46 Changes to SUNDIALS in release 2.6.1

Fixed loop limit bug in `SlsAddMat` function.

In all six solver interfaces to KLU and SuperLUMT, added `#include` lines, and removed redundant KLU structure allocations.

Minor bug fixes in ARKODE.

## 23.47 Changes to SUNDIALS in release 2.6.0

### Autotools Build Option Removed

With this version of SUNDIALS, support and documentation of the Autotools mode of installation is being dropped, in favor of the CMake mode, which is considered more widely portable.

### New Package: ARKODE

Addition of ARKODE package of explicit, implicit, and additive Runge-Kutta methods for ODEs. This package API is close to CVODE so switching between the two should be straightforward. Thanks go to Daniel Reynolds for the addition of this package.

### New Features and Enhancements

Added *OpenMP* and *Pthreads N\_Vector* implementations for thread-parallel computing environments.



Two major additions were made to the linear system solvers available in all packages. First, in the serial case, an interface to the sparse direct solver KLU was added. Second, an interface to SuperLU\_MT, the multi-threaded version of SuperLU, was added as a thread-parallel sparse direct solver option, to be used with the serial version of the N\_Vector module. As part of these additions, a sparse matrix (CSC format) structure was added to CVODE.

### *KINSOL*

Two major additions were made to the globalization strategy options (KINSol argument `strategy`). One is fixed-point iteration, and the other is Picard iteration. Both can be accelerated by use of the Anderson acceleration method. See the relevant paragraphs in Chapter [Mathematical Considerations](#).

An interface to the Flexible GMRES iterative linear solver was added.

### **Bug Fixes**

In order to avoid possible name conflicts, the mathematical macro and function names MIN, MAX, SQR, RAbs, RSqrt, RExp, RPowerI, and RPowerR were changed to SUNMIN, SUNMAX, SUNSQR, SUNRabs, SUNRsqr, SUNRexp, SRpowerI, and SUNRpowerR, respectively. These names occur in both the solver and example programs.

In the LAPACK banded linear solver interfaces, the line `smu = MIN(N-1,mu+ml)` was changed to `smu = mu + ml` to correct an illegal input error for to DGBTRF and DGBTRS.

In all Fortran examples, integer declarations were revised so that those which must match a C type `long int` are declared `INTEGER*8`, and a comment was added about the type match. All other integer declarations are just `INTEGER`. Corresponding minor corrections were made to the user guide.

### *CVODE and CVODES*

In `cvRootFind`, a minor bug was corrected, where the input array was ignored, and a line was added to break out of root-search loop if the initial interval size is below the tolerance `ttol`.

Two minor bugs were fixed regarding the testing of input on the first call to `CVode` – one involving `tstop` and one involving the initialization of `*tret`.

The example program `cvAdvDiff_diag_p` was added to illustrate the use of `in_parallel`.

In the FCVODE optional input routines `FCVSETIIN` and `FCVSETRIN`, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all tests.

In order to eliminate or minimize the differences between the sources for private functions in CVODE and CVODES, the names of many private functions were changed from `CV*` to `cv*` and a few other names were also changed.

An option was added in the case of Adjoint Sensitivity Analysis with dense or banded Jacobian. With a call to `CVDlsSetDenseJacFnBS` or `CVDlsSetBandJacFnBS`, the user can specify a user-supplied Jacobian function of type `CVDls***JacFnBS`, for the case where the backward problem depends on the forward sensitivities.

In `CVodeQuadSensInit`, the line `cv_mem->cv_fQS_data = ...` was corrected (missing Q).

In the CVODES User Guide, a paragraph was added in Section 6.2.1 on `CVodeAdjReInit`, and a paragraph was added in Section 6.2.9 on `CVodeGetAdjY`. In the example `cvsRoberts_ASAi_dns`, the output was revised to include the use of `CVodeGetAdjY`.

For the Adjoint Sensitivity Analysis case in which the backward problem depends on the forward sensitivities, options have been added to allow for user-supplied `pset`, `psolve`, and `jtimes` functions.

In the example `cvsHessian_ASA_FSA`, an error was corrected in the function `fB2`, `y2` in place of `y3` in the third term of `Ith(yBdot,6)`.

### *IDA and IDAS*

In `IDARootfind`, a minor bug was corrected, where the input array `rootdir` was ignored, and a line was added to break out of root-search loop if the initial interval size is below the tolerance `ttol`.

A minor bug was fixed regarding the testing of the input `tstop` on the first call to `IDASolve()`.



In the FIDA optional input routines FIDASETIIN, FIDASETRIN, and FIDASETVIN, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strcmp` tests.

An option was added in the case of Adjoint Sensitivity Analysis with dense or banded Jacobian. With a call to `IDADlsSetDenseJacFnBS` or `IDADlsSetBandJacFnBS`, the user can specify a user-supplied Jacobian function of type `IDADls***JacFnBS`, for the case where the backward problem depends on the forward sensitivities.

#### *KINSOL*

In function `KINStop`, two return values were corrected to make the values of `uu` and `fval` consistent.

A bug involving initialization of `mxnewtstep` was fixed. The error affects the case of repeated user calls to `KINSOL` with no intervening call to `KINSetMaxNewtonStep`.

A bug in the increments for difference quotient Jacobian approximations was fixed in function `kinDlsBandDQJac`.

In the FKINSOL module, an incorrect return value `ier` in `FKINfunc` was fixed.

In the FKINSOL optional input routines FKINSETIIN, FKINSETRIN, and FKINSETVIN, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strcmp` tests.

## 23.48 Changes to SUNDIALS in release 2.5.0

### Integer Type Change

One significant design change was made with this release, the problem size and its relatives, bandwidth parameters, related internal indices, pivot arrays, and the optional output `lsflag` have all been changed from type `int` to type `long int`, except for the problem size and bandwidths in user calls to routines specifying BLAS/LAPACK routines for the dense/band linear solvers. The function `NewIntArray` is replaced by a pair `NewIntArray / NewLintArray`, for `int` and `long int` arrays, respectively.

### Bug Fixes

In the installation files, we modified the treatment of the macro `SUNDIALS_USE_GENERIC_MATH`, so that the parameter `GENERIC_MATH_LIB` is either defined (with no value) or not defined.

In all packages, after the solver memory is created, it is set to zero before being filled.

In each linear solver interface function, the linear solver memory is freed on an error return, and the function now includes a line setting to NULL the main memory pointer to the linear solver memory.

#### *Rootfinding*

In `CVODE(S)` and `IDA(S)`, in the functions `Rcheck1` and `Rcheck2`, when an exact zero is found, the array `glo` of  $g$  values at the left endpoint is adjusted, instead of shifting the  $t$  location `tlo` slightly.

#### *CVODE and CVODES*

In `CVSetTqBDF`, the logic was changed to avoid a divide by zero.

In a minor change to the CVODES user interface, the type of the index `which` was changed from `long int` to `int`.

Errors in the logic for the integration of backward problems in CVODES were identified and fixed.

#### *IDA and IDAS*

To be consistent with IDAS, IDA uses the function `IDAGetDky` for optional output retrieval.

A memory leak was fixed in two of the `IDASp***Free` functions.

A missing vector pointer setting was added in `IDASensLineSrch`.

In `IDACompleteStep`, conditionals around lines loading a new column of three auxiliary divided difference arrays, for a possible order increase, were fixed.

### KINSOL

Three major logic bugs were fixed - involving updating the solution vector, updating the linesearch parameter, and a missing error return.

Three minor errors were fixed - involving setting `etachoice` in the Matlab/KINSOL interface, a missing error case in `KINPrintInfo`, and avoiding an exponential overflow in the evaluation of `omega`.

## 23.49 Changes to SUNDIALS in release 2.4.0

Added a CMake-based build option in addition to the one based on autotools.

The user interface has been further refined. Some of the API changes involve:

- (a) a reorganization of all linear solver modules into two families (besides the existing family of scaled preconditioned iterative linear solvers, the direct solvers, including new LAPACK-based ones, were also organized into a *direct* family);
- (b) maintaining a single pointer to user data, optionally specified through a Set-type function; and
- (c) a general streamlining of the preconditioner modules distributed with the solvers.

Added interfaces to LAPACK linear solvers for dense and banded matrices to all packages.

An option was added to specify which direction of zero-crossing is to be monitored while performing rootfinding in CVODE(S) and IDA(S).

CVODES includes several new features related to sensitivity analysis, among which are:

- (a) support for integration of quadrature equations depending on both the states and forward sensitivity (and thus support for forward sensitivity analysis of quadrature equations),
- (b) support for simultaneous integration of multiple backward problems based on the same underlying ODE (e.g., for use in an *forward-over-adjoint* method for computing second order derivative information),
- (c) support for backward integration of ODEs and quadratures depending on both forward states and sensitivities (e.g., for use in computing second-order derivative information), and
- (d) support for reinitialization of the adjoint module.

Moreover, the prototypes of all functions related to integration of backward problems were modified to support the simultaneous integration of multiple problems.

All backward problems defined by the user are internally managed through a linked list and identified in the user interface through a unique identifier.

## 23.50 Changes to SUNDIALS in release 2.3.0

### New Features and Enhancements

The main changes in this release involve a rearrangement of the entire SUNDIALS source tree. At the user interface level, the main impact is in the mechanism of including SUNDIALS header files which must now include the relative path e.g., `#include <cvcde/cvcde.h>` as all exported header files are now installed in separate subdirectories of the installation *include* directory.

The functions in the generic dense linear solver (`sundials_dense` and `sundials_smalldense`) were modified to work for rectangular  $m \times n$  matrices ( $m \leq n$ ), while the factorization and solution functions were renamed to `DenseGETRF` / `denGETRF` and `DenseGETRS` / `denGETRS`, respectively. The factorization and solution functions in the generic band linear solver were renamed `BandGBTRF` and `BandGBTRS`, respectively.

In IDA, the user interface to the consistent initial conditions calculations was modified. The `IDACalcIC()` arguments `t0`, `yy0`, and `yp0` were removed and a new function, `IDAGetConsistentIC()` is provided.

### Bug Fixes

In the CVODES adjoint solver module, the following two bugs were fixed:

- In `CVodeF` the solver was sometimes incorrectly taking an additional step before returning control to the user (in `CV_NORMAL` mode) thus leading to a failure in the interpolated output function.
- In `CVodeB`, while searching for the current check point, the solver was sometimes reaching outside the integration interval resulting in a segmentation fault.

In IDA, a bug was fixed in the internal difference-quotient dense and banded Jacobian approximations, related to the estimation of the perturbation (which could have led to a failure of the linear solver when zero components with sufficiently small absolute tolerances were present).

## 23.51 Changes to SUNDIALS in release 2.2.0

### New Header Files Names

To reduce the possibility of conflicts, the names of all header files have been changed by adding unique prefixes (e.g., `cvode_` and `sundials_`). When using the default installation procedure, the header files are exported under various subdirectories of the target `include` directory. For more details see Appendix §17.

### Build System Changes

Updated configure script and Makefiles for Fortran examples to avoid C++ compiler errors (now use `CC` and `MPICC` to link only if necessary).

The shared object files are now linked into each SUNDIALS library rather than into a separate `libsundials_shared` library.

### New Features and Enhancements

Deallocation functions now take the address of the respective memory block pointer as the input argument.

Interfaces to the Scaled Preconditioned Bi-CGstab (SPBCG) and Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR) linear solver modules have been added to all packages. At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions. Additionally, in KINSOL interfaces have been added to the SUNDIALS DENSE, and BAND linear solvers and include support for nonlinear residual monitoring which can be used to control Jacobian updating.

A new interpolation method was added to the CVODES adjoint module. The function `CVadjMalloc` has an additional argument which can be used to select the desired interpolation scheme.

FIDA, a Fortran-C interface module, was added.

The rootfinding feature was added to IDA, whereby the roots of a set of given functions may be computed during the integration of the DAE system.

In IDA a user-callable routine was added to access the estimated local error vector.

In the KINSOL Fortran interface module, FKINSOL, optional inputs are now set using `FKINSETIIN` (integer inputs), `FKINSETRIN` (real inputs), and `FKINSETVIN` (vector inputs). Optional outputs are still obtained from the `IOUT` and `ROUT` arrays which are owned by the user and passed as arguments to `FKINMALLOC`.

## 23.52 Changes to SUNDIALS in release 2.1.1

The function `N_VCloneEmpty` was added to the global vector operations table.

A minor bug was fixed in the interpolation functions of the adjoint CVODES module.

## 23.53 Changes to SUNDIALS in release 2.1.0

The user interface has been further refined. Several functions used for setting optional inputs were combined into a single one.

In CVODE(S) and IDA, an optional user-supplied routine for setting the error weight vector was added.

Additionally, to resolve potential variable scope issues, all SUNDIALS solvers release user data right after its use.

The build systems has been further improved to make it more robust.

## 23.54 Changes to SUNDIALS in release 2.0.2

Fixed autoconf-related bug to allow configuration with the PGI Fortran compiler.

Modified the build system to use customized detection of the Fortran name mangling scheme (autoconf's `AC_F77_WRAPPERS` routine is problematic on some platforms).

A bug was fixed in the `CVode` function that was potentially leading to erroneous behavior of the rootfinding procedure on the integration first step.

A new chapter in the User Guide was added - with constants that appear in the user interface.

## 23.55 Changes to SUNDIALS in release 2.0.1

### Build System

Changed the order of compiler directives in header files to avoid compilation errors when using a C++ compiler.

Changed the method of generating `sundials_config.h` to avoid potential warnings of redefinition of preprocessor symbols.

### New Features

In CVODES the option of activating and deactivating forward sensitivity calculations on successive runs without memory allocation and deallocation.

### Bug Fixes

In CVODES bug fixes related to forward sensitivity computations (possible loss of accuracy on a BDF order increase and incorrect logic in testing user-supplied absolute tolerances) were made.

## 23.56 Changes to SUNDIALS in release 2.0.0

Installation of all of SUNDIALS packages has been completely redesigned and is now based on configure scripts.

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver

through the `iopt` and `ropt` arrays. Instead, packages now provide `Set` functions to change the default values for various quantities controlling the solver and `Get` functions to extract statistics after return from the main solver routine.

Additionally, the interfaces to several user-supplied routines (such as those providing Jacobians and preconditioner information) were simplified by reducing the number of arguments. The same information that was previously accessible through such arguments can now be obtained through `Get`-type functions.

In CVODE and CVODES a rootfinding feature was added, whereby the roots of a set of given functions may be computed during the integration of the ODE system.

Changes to the `NVector`:

- Removed `machEnv`, redefined table of vector operations (now contained in the `N_Vector` structure itself).
- All SUNDIALS functions create new `N_Vector` variables through cloning, using an `N_Vector` passed by the user as a template.
- A particular vector implementation is supposed to provide user-callable constructor and destructor functions.
- Removed the following functions from the structure of vector operations: `N_VNew`, `N_VNew_S`, `N_VFree`, `N_VFree_S`, `N_VMake`, `N_VDispose`, `N_VGetData`, `N_VSetData`, `N_VConstrProdPos`, and `N_VOneMask`.
- Added the following functions to the structure of vector operations: `N_VClone`, `N_VDestroy`, `N_VSpace`, `N_VGetArrayPointer`, `N_VSetArrayPointer`, and `N_VWrmsNormMask`.
- Note that `nvec_ser` and `nvec_par` are now separate modules outside the shared SUNDIALS module.

Changes to the linear solvers:

- In SPGMR, added a dummy `N_Vector` argument to be used as a template for cloning.
- In SPGMR, removed `N` (problem dimension) from the argument list of `SpgmrMalloc`.
- Replaced `iterativ.{c,h}` with `iterative.{c,h}`.
- Modified constant names in `iterative.h` (preconditioner types are prefixed with `PREC_`).
- Changed numerical values for `MODIFIED_GS` (from 0 to 1) and `CLASSICAL_GS` (from 1 to 2).

Changes to `sundialsmath` submodule:

- Replaced the internal routine for estimating unit roundoff with definition of unit roundoff from `float.h`.
- Modified functions to call the appropriate math routines given the precision level specified by the user.

Changes to `sundialstypes` submodule:

- Removed `integertype`.
- Added definitions for `BIG_REAL`, `SMALL_REAL`, and `UNIT_ROUNDOFF` using values from `float.h` based on the precision.
- Changed definition of macro `RCONST` to depend on the precision level specified by the user.



# Bibliography

- [1] Xbraid: parallel multigrid in time. <http://llnl.gov/casc/xbraid>.
- [2] AMD ROCm Documentation. <https://rocmdocs.amd.com/en/latest/index.html>.
- [3] Intel oneAPI Programming Guide. <https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html>.
- [4] KLU Sparse Matrix Factorization Library. <http://faculty.cse.tamu.edu/davis/suitesparse.html>.
- [5] NVIDIA CUDA Programming Guide. <https://docs.nvidia.com/cuda/index.html>.
- [6] NVIDIA cuSOLVER Programming Guide. <https://docs.nvidia.com/cuda/cusolver/index.html>.
- [7] NVIDIA cuSPARSE Programming Guide. <https://docs.nvidia.com/cuda/cusparse/index.html>.
- [8] SuperLU\_DIST Parallel Sparse Matrix Factorization Library. [https://portal.nersc.gov/project/sparse/superlu/#superlu\\_dist](https://portal.nersc.gov/project/sparse/superlu/#superlu_dist).
- [9] SuperLU\_MT Threaded Sparse Matrix Factorization Library. [https://portal.nersc.gov/project/sparse/superlu/#superlu\\_mt](https://portal.nersc.gov/project/sparse/superlu/#superlu_mt).
- [10] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965. doi:10.1145/321296.321305.
- [11] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Transactions on Mathematical Software*, 48(1):2:1–2:33, February 2022. URL: 10.1145/3480935 (visited on 2022-02-17), doi:10.1145/3480935.
- [12] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9(1):17–29, 1951. doi:10.1090/qam/42792.
- [13] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia, Pa, 1998.
- [14] Winfried Auzinger, Harald Hofstätter, David Ketcheson, and Othmar Koch. Practical splitting methods for the adaptive integration of nonlinear evolution equations. Part I: Construction of optimized schemes and pairs of schemes. *BIT Numerical Mathematics*, 57(1):55–74, July 2016. doi:10.1007/s10543-016-0626-9.
- [15] Cody J Balos, David J Gardner, Carol S Woodward, and Daniel R Reynolds. Enabling GPU accelerated computing in the SUNDIALS time integration library. *Parallel Computing*, 108:102836, 2021. doi:10.1016/j.parco.2021.102836.
- [16] R.E. Bank, W.M. Coughran, W. Fichtner, E.H. Grosse, D.J. Rose, and R.K. Smith. Transient simulation of silicon devices and circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4(4):436–451, 1985. doi:10.1109/TCAD.1985.1270142.
- [17] S.R Billington. Type-insensitive codes for the solution of stiff and nonstiff systems of ordinary differential equations. *Master Thesis, University of Manchester, United Kingdom*, 1983.

- [18] Sergio Blanes, Fernando Casas, and Ander Murua. Splitting methods for differential equations. *Acta Numerica*, 33:1–161, 2024. doi:10.1017/S0962492923000077.
- [19] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: performance introspection for hpc software stacks. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 550–560. IEEE, 2016. doi:10.1109/SC.2016.46.
- [20] P. Bogacki and L.F. Shampine. A 3(2) pair of Runge-Kutta formulas. *Applied Mathematics Letters*, 2(4):321–325, 1989. doi:10.1016/0893-9659(89)90079-7.
- [21] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987. doi:10.1137/0724031.
- [22] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. VODE, a Variable-Coefficient ODE Solver. *SIAM J. Sci. Stat. Comput.*, 10:1038–1051, 1989. doi:10.1137/0910062.
- [23] P. N. Brown and A. C. Hindmarsh. Reduced storage matrix methods in stiff ODE systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989. doi:10.1016/0096-3003(89)90110-0.
- [24] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990. doi:10.1137/0911026.
- [25] J.C. Butcher. *Numerical Methods for Ordinary Differential Equations*. Wiley, Chichester, England, 2 edition, 2008.
- [26] 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*, 323–356. Oxford, 1992. Oxford University Press.
- [27] J Candy and W Rozmus. A symplectic integration algorithm for separable hamiltonian functions. *Journal of Computational Physics*, 92(1):230–256, 1991.
- [28] J.R. Cash. Diagonally Implicit Runge-Kutta Formulae with Error Estimates. *IMA Journal of Applied Mathematics*, 24(3):293–301, 1979. doi:10.1093/imamat/24.3.293.
- [29] J.R. Cash and A.H. Karp. A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides. *ACM Transactions on Mathematical Software*, 16(3):201–222, 1990. doi:10.1145/79505.79507.
- [30] Rujeko Chinomona and Daniel R Reynolds. Implicit-explicit multirate infinitesimal GARK methods. *SIAM Journal on Scientific Computing*, 43(5):A3082–A3113, 2021. doi:10.1137/20M1354349.
- [31] Michael Creutz and Andreas Gocksch. Higher-order hybrid monte carlo algorithms. *Phys. Rev. Lett.*, 63:9–12, Jul 1989. doi:10.1103/PhysRevLett.63.9.
- [32] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3):1–17, 2010. doi:10.1145/1824801.1824814.
- [33] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19(2):400–408, 1982. doi:10.1137/0719025.
- [34] 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(4):915–952, 1999. doi:10.1137/S0895479897317685.
- [35] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996. doi:10.1137/1.9781611971200.
- [36] Fasma Diele and Carmela Marangi. Explicit symplectic partitioned Runge-Kutta-Nyström methods for non-autonomous dynamics. *Applied Numerical Mathematics*, 61(7):832–843, 2011. doi:10.1016/j.apnum.2011.02.003.



- [37] J.R. Dormand and P.J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980. doi:10.1016/0771-050X(80)90013-3.
- [38] M.R. Dorr, J.-L. Fattebert, M.E. Wickett, J.F. Belak, and P.E.A. Turchi. A numerical algorithm for the solution of a phase-field model of polycrystalline materials. *Journal of Computational Physics*, 229(3):626–641, 2010. doi:10.1016/j.jcp.2009.09.041.
- [39] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. doi:10.1016/j.jpdc.2014.07.003.
- [40] Leonhard Euler. *Institutiones calculi integralis*. Volume Volumen Primum. B. G. Teubner Verlag, 1768. reprinted in Opera Omnia Series 1, Volume 11.
- [41] R.D. Falgout, S. Friedhoff, TZ.V. Kolev, S.P. MacLachlan, and J.B. Schroder. Parallel time integration with multigrid. *SIAM Journal of Scientific Computing*, 36(6):C635–C661, 2014. doi:10.1137/130944230.
- [42] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16(3):197–221, 2009. doi:10.1002/nla.617.
- [43] E. Fehlberg. Low-order classical runge-kutta formulas with step size control and their application to some heat transfer problems. Technical Report 315, NASA, 1969.
- [44] Imre Fekete, Sidafa Conde, and John N. Shadid. Embedded pairs for optimal explicit strong stability preserving Runge–Kutta methods. *Journal of Computational and Applied Mathematics*, 412:114325, 2022. doi:10.1016/j.cam.2022.114325.
- [45] Imre Fekete, Sidafa Conde, and John N. Shadid. Embedded pairs for optimal explicit strong stability preserving Runge–Kutta methods. *Journal of Computational and Applied Mathematics*, 412:114325, 2022. doi:10.1016/j.cam.2022.114325.
- [46] Alex C. Fish, Daniel R. Reynolds, and Steven B. Roberts. Implicit–explicit multirate infinitesimal stage-restart methods. *Journal of Computational and Applied Mathematics*, 438:115534, 2024. doi:10.1016/j.cam.2023.115534.
- [47] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14(2):470–482, 1993. doi:10.1137/0914029.
- [48] Amir Gholami, Kurt Keutzer, and George Biros. Anode: unconditionally accurate memory-efficient gradients for neural odes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 730–736, 2019.
- [49] Michael B Giles and Niles A Pierce. An introduction to the adjoint approach to design. *Flow, turbulence and combustion*, 65(3):393–415, 2000.
- [50] F. X. Giraldo, J. F. Kelly, and E. M. Constantinescu. Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (numa). *SIAM Journal on Scientific Computing*, 35(5):B1162–B1194, 2013. doi:10.1137/120876034.
- [51] Laura Grigori, James W. Demmel, and Xiaoye S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Scientific Computing*, 29(3):1289–1314, 2007. doi:10.1137/050638102.
- [52] K. Gustafsson. Control theoretic techniques for stepsize selection in explicit Runge-Kutta methods. *ACM Transactions on Mathematical Software*, 17(4):533–554, 1991. doi:10.1145/210232.210242.
- [53] K. Gustafsson. Control theoretic techniques for stepsize selection in implicit Runge-Kutta methods. *ACM Transactions on Mathematical Software*, 20(4):496–512, 1994. doi:10.1145/198429.198437.
- [54] William W Hager. Runge-Kutta methods in optimal control and the transformed adjoint system. *Numerische Mathematik*, 87:247–282, 2000. doi:10.1007/s002110000178.
- [55] E. Hairer, S. P. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I*. Springer-Verlag, Berlin, 1987.

- [56] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems*. Springer-Verlag, Berlin, 1991.
- [57] Ernst Hairer, Gerhard Wanner, and Christian Lubich. *Geometric Numerical Integration, Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer Series in Computational Mathematics, 2006. doi:[10.1007/3-540-30666-8](https://doi.org/10.1007/3-540-30666-8).
- [58] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Research of the National Bureau of Standards*, 49(6):409–436, 1952. doi:[10.6028/jres.049.044](https://doi.org/10.6028/jres.049.044).
- [59] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.
- [60] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, Dec 2000.
- [61] A. C. Hindmarsh and A. G. Taylor. PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-ID-129739, LLNL, February 1998.
- [62] Alan C. Hindmarsh and Radu Serban. Example Programs for CVODE v7.7.0. Technical Report, LLNL, 2026. UCRL-SM-208110.
- [63] TE Hull, WH Enright, and KR Jackson. User's guide for DVERK: A subroutine for solving non-stiff ODE's. Technical Report 100, University of Toronto. Department of Computer Science, 1976.
- [64] Laurent O Jay. Symplecticness conditions of some low order partitioned methods for non-autonomous hamiltonian systems. *Numerical Algorithms*, 86(2):495–514, 2021.
- [65] Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. Automated fortran-c++ bindings for large-scale scientific applications. *Computing in Science & Engineering*, 22(5):84–94, 2020. doi:[10.1109/MCSE.2019.2924204](https://doi.org/10.1109/MCSE.2019.2924204).
- [66] Shinhoo Kang and Emil M Constantinescu. Entropy–Preserving and Entropy–Stable Relaxation IMEX and Multirate Time–Stepping Methods. *Journal of Scientific Computing*, 93(1):1–31, 2022. doi:[10.1007/s10915-022-01982-w](https://doi.org/10.1007/s10915-022-01982-w).
- [67] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995. doi:[10.1137/1.9781611970944](https://doi.org/10.1137/1.9781611970944).
- [68] C.A. Kennedy and M.H. Carpenter. Additive runge-kutta schemes for convection-diffusion-reaction equations. *Applied Numerical Mathematics*, 44(1-2):139–181, 2003. doi:[10.1016/S0168-9274\(02\)00138-1](https://doi.org/10.1016/S0168-9274(02)00138-1).
- [69] C.A. Kennedy and M.H. Carpenter. Diagonally implicit Runge–Kutta methods for ordinary differential equations. a review. Technical Report TM-2016-219173, NASA, 2016.
- [70] C.A. Kennedy and M.H. Carpenter. Diagonally implicit Runge–Kutta methods for stiff ODEs. *Applied Numerical Mathematics*, 146():221–244, 2019. doi:[10.1016/j.apnum.2019.07.008](https://doi.org/10.1016/j.apnum.2019.07.008).
- [71] C.A. Kennedy and M.H. Carpenter. Higher-order additive runge-kutta schemes for ordinary differential equations. *Applied Numerical Mathematics*, 136:183–205, 2019. doi:[10.1016/j.apnum.2018.10.007](https://doi.org/10.1016/j.apnum.2018.10.007).
- [72] David I Ketcheson. Relaxation Runge–Kutta methods: Conservation and stability for inner-product norms. *SIAM Journal on Numerical Analysis*, 57(6):2850–2870, 2019. doi:[10.1137/19M1263662](https://doi.org/10.1137/19M1263662).
- [73] David I. Ketcheson. Highly efficient strong stability-preserving Runge–Kutta methods with low-storage implementations. *SIAM Journal on Scientific Computing*, 30(4):2113–2136, 2008. doi:[10.1137/07070485X](https://doi.org/10.1137/07070485X).
- [74] O. Knuth and R. Wolke. Implicit-explicit runge–kutta methods for computing atmospheric reactive flows. *Applied Numerical Analysis*, 28(2-4):327–341, 1998. doi:[10.1016/S0168-9274\(98\)00051-8](https://doi.org/10.1016/S0168-9274(98)00051-8).
- [75] A. Kværno. Singly Diagonally Implicit Runge-Kutta Methods with an Explicit First Stage. *BIT Numerical Mathematics*, 44:489–502, 2004. doi:[10.1023/B:BITN.0000046811.70614.38](https://doi.org/10.1023/B:BITN.0000046811.70614.38).
- [76] X. S. Li. An overview of SuperLU: algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005. doi:[10.1145/1089014.1089017](https://doi.org/10.1145/1089014.1089017).

- [77] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. <http://crd.lbl.gov/\protect\unhbox\voidb@x\penalty\@M\xiaoye/SuperLU/>. Last update: August 2011.
- [78] Xiaoye S. Li and James W. Demmel. SuperLU\_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003. doi:10.1145/779359.779361.
- [79] P. A. Lott, H. F. Walker, C. S. Woodward, and U. M. Yang. An accelerated Picard method for nonlinear systems related to variably saturated flow. *Adv. Wat. Resour.*, 38:92–101, 2012. doi:10.1016/j.advwatres.2011.12.013.
- [80] Vu Thai Luan, Rujeko Chinomona, and Daniel R. Reynolds. A new class of high-order methods for multirate differential equations. *SIAM Journal of Scientific Computing*, 42(2):A1245–A1268, 2020. doi:10.1137/19M125621X.
- [81] Robert I Mclachlan and Pau Atela. The accuracy of symplectic integrators. *Nonlinearity*, 5(2):541, 1992.
- [82] Chad D. Meyer, Dinshaw S. Balsara, and Tariq D. Aslam. A stabilized Runge–Kutta–Legendre method for explicit super-time-stepping of parabolic and mixed equations. *Journal of Computational Physics*, 257:594–626, 2014. doi:10.1016/j.jcp.2013.08.021.
- [83] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.
- [84] Anthony Ralston. Runge–Kutta methods with minimum error bounds. *Mathematics of Computation*, 16(80):431–437, 1962. doi:10.1090/s0025-5718-1962-0150954-0.
- [85] Hendrik Ranocha and David I Ketcheson. Relaxation Runge–Kutta methods for Hamiltonian problems. *Journal of Scientific Computing*, 84(1):1–27, 2020. doi:10.1007/s10915-020-01277-y.
- [86] Hendrik Ranocha, Mohammed Sayyari, Lisandro Dalcin, Matteo Parsani, and David I Ketcheson. Relaxation Runge–Kutta Methods: Fully Discrete Explicit Entropy-Stable Schemes for the Compressible Euler and Navier–Stokes Equations. *SIAM Journal on Scientific Computing*, 42(2):A612–A638, 2020. doi:10.1137/19M1263480.
- [87] Daniel R. Reynolds. Example Programs for ARKODE v6.7.0. Technical Report, Southern Methodist University, 2026.
- [88] Steven Roberts, Andrey A Popov, Arash Sarshar, and Adrian Sandu. A fast time-stepping strategy for dynamical systems equipped with a surrogate model. *SIAM Journal on Scientific Computing*, 44(3):A1405–A1427, 2022. doi:10.1137/20M1386281.
- [89] C. Runge. Ueber die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895. doi:10.1007/BF01446807.
- [90] Ronald D Ruth. A canonical integration technique. *IEEE Trans. Nucl. Sci.*, 30(CERN-LEP-TH-83-14):2669–2671, 1983.
- [91] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993. doi:10.1137/0914028.
- [92] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7(3):856–869, 1986. doi:10.1137/0907058.
- [93] A. Sandu. A class of multirate infinitesimal gark methods. *SIAM Journal of Numerical Analysis*, 57(5):2300–2327, 2019. doi:10.1137/18M1205492.
- [94] Adrian Sandu. On the properties of Runge-Kutta discrete adjoints. *Lecture Notes in Computer Science*, pages 550–557, 2006. doi:10.1007/11758549\_76.
- [95] A. Sayfy and A. Aburub. Embedded Additive Runge-Kutta Methods. *International Journal of Computer Mathematics*, 79(8):945–953, 2002. doi:10.1080/00207160212109.

- [96] M. Schlegel, O. Knoth, M. Arnold, and R. Wolke. Multirate Runge–Kutta schemes for advection equations. *Journal of Computational Applied Mathematics*, 226(2):345–357, 2009. doi:10.1016/j.cam.2008.08.009.
- [97] M. Schlegel, O. Knoth, M. Arnold, and R. Wolke. Implementation of multirate time integration methods for air pollution modelling. *GMD*, 5(6):1395–1405, 2012. doi:10.5194/gmd-5-1395-2012.
- [98] M. Schlegel, O. Knoth, M. Arnold, and R. Wolke. Numerical solution of multiscale problems in atmospheric modeling. *Applied Numerical Mathematics*, 62(10):1531–1542, 2012. doi:10.1016/j.apnum.2012.06.023.
- [99] L. F. Shampine. Implementation of implicit formulas for the solution of ODEs. *SIAM Journal on Scientific and Statistical Computing*, 1(1):103–118, 1980. doi:10.1137/0901005.
- [100] Chi-Wang Shu and Stanley Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of computational physics*, 77(2):439–471, 1988. doi:10.1016/0021-9991(88)90177-5.
- [101] Chi-Wang Shu and Stanley Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of Computational Physics*, 77(2):439–471, 1988. doi:10.1016/0021-9991(88)90177-5.
- [102] Ziv Sirkes and Eli Tziperman. Finite difference of adjoint or adjoint of finite difference? *Monthly weather review*, 125(12):3373–3378, 1997.
- [103] G. Soderlind. The automatic control of numerical integration. *CWI Quarterly*, 11:55–74, 1998.
- [104] G. Soderlind. Digital filters in adaptive time-stepping. *ACM Transactions on Mathematical Software*, 29(1):1–26, 2003. doi:10.1145/641876.641877.
- [105] G. Soderlind. Time-step selection algorithms: Adaptivity, control and signal processing. *Applied Numerical Mathematics*, 56(3-4):488–502, 2006. doi:10.1016/j.apnum.2005.04.026.
- [106] M. Sofroniou and G. Spaletta. Construction of explicit Runge–Kutta pairs with stiffness detection. *Mathematical and Computer Modelling*, 40(11):1157–1169, 2004. doi:10.1016/j.mcm.2005.01.010.
- [107] Mark Sofroniou and Giulia Spaletta. Symplectic Methods for Separable Hamiltonian Systems. *Lecture Notes in Computer Science*, pages 506–515, 2002. doi:10.1007/3-540-47789-6\_53.
- [108] Mark Sofroniou and Giulia Spaletta. Derivation of symmetric composition constants for symmetric integrators. *Optimization Methods and Software*, 20(4-5):597–613, 2005.
- [109] Raymond J Spiteri and Steven J Ruuth. A new class of optimal high-order strong-stability-preserving time discretization methods. *SIAM Journal on Numerical Analysis*, 40(2):469–491, 2002. doi:10.1137/S0036142901389025.
- [110] Gilbert Strang. Accurate partial difference methods I: Linear cauchy problems. *Archive for Rational Mechanics and Analysis*, 12(1):392–402, January 1963. doi:10.1007/bf00281235.
- [111] Gilbert Strang. On the construction and comparison of difference schemes. *SIAM Journal on Numerical Analysis*, 5(3):506–517, September 1968. doi:10.1137/0705041.
- [112] Jürgen Struckmeier and Claus Riedel. Canonical transformations and exact invariants for time-dependent hamiltonian systems. *Annalen der Physik*, 11(1):15–38, 2002.
- [113] M Suzuki and K Umeno. Higher-order decomposition theory of exponential operators and its applications to qmc and nonlinear dynamics. *Computer simulation studies in condensed-matter physics VI*, pages 74–86, 1993.
- [114] Masuo Suzuki. Fractal decomposition of exponential operators with applications to many-body theories and Monte Carlo simulations. *Physics Letters A*, 146(6):319–323, June 1990. doi:10.1016/0375-9601(90)90962-n.
- [115] Masuo Suzuki. General nonsymmetric higher-order decomposition of exponential operators and symplectic integrators. *Journal of the Physical Society of Japan*, 61(9):3015–3019, September 1992. doi:10.1143/jpsj.61.3015.
- [116] Molei Tao and Shi Jin. Accurate and efficient simulations of hamiltonian mechanical systems with discontinuous potentials. *Journal of Computational Physics*, 450:110846, 2022. URL: <https://www.sciencedirect.com/science/article/pii/S0021999121007415>, doi:10.1016/j.jcp.2021.110846.

- [117] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010. doi:[10.1016/j.parco.2009.12.005](https://doi.org/10.1016/j.parco.2009.12.005).
- [118] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. The kokkos ecosystem: comprehensive performance portability for high performance computing. *Computing in Science Engineering*, 23(5):10–18, 2021. doi:[10.1109/MCSE.2021.3098509](https://doi.org/10.1109/MCSE.2021.3098509).
- [119] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022. doi:[10.1109/TPDS.2021.3097283](https://doi.org/10.1109/TPDS.2021.3097283).
- [120] Ch. Tsitouras. Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, 62(2):770–775, July 2011. doi:[10.1016/j.camwa.2011.06.002](https://doi.org/10.1016/j.camwa.2011.06.002).
- [121] J.H Verner. Numerically optimal Runge–Kutta pairs with interpolants. *Numerical Algorithms*, 53(2):383–396, 2010. doi:[10.1007/s11075-009-9290-3](https://doi.org/10.1007/s11075-009-9290-3).
- [122] J.G. Verwer, B.P. Sommeijer, and W. Hundsdorfer. RKC time-stepping for advection–diffusion–reaction problems. *Journal of Computational Physics*, 201(1):61–79, 2004. doi:[10.1016/j.jcp.2004.05.002](https://doi.org/10.1016/j.jcp.2004.05.002).
- [123] Richard von Mises and Hilda Pollaczek-Geiringer. Praktische verfahren der gleichungsauflösung. *Zeitschrift für Angewandte Mathematik und Mechanik*, 9:152–164, 1929. doi:[10.1002/zamm.19290090206](https://doi.org/10.1002/zamm.19290090206).
- [124] 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(2):631–644, 1992. doi:[10.1137/0913035](https://doi.org/10.1137/0913035).
- [125] H. F. Walker and P. Ni. Anderson Acceleration for Fixed-Point Iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011. doi:[10.1137/10078356X](https://doi.org/10.1137/10078356X).
- [126] Haruo Yoshida. Construction of higher order symplectic integrators. *Physics letters A*, 150(5-7):262–268, 1990.
- [127] J.A. Zonneveld. Automatic integration of ordinary differential equations. Technical Report R743, Mathematisch Centrum, Postbus 4079, 1009AB Amsterdam, 1963.



# Python Module Index

## S

`sundials4py.arkode`, [878](#)  
`sundials4py.core`, [848](#)





# Index

## Symbols

- `_MRIStepInnerStepper` (class in `sundials4py.arkode`), 883
- `_N_VectorContent_ManyVector` (class in `sundials4py.core`), 852
- `_N_VectorContent_Serial` (class in `sundials4py.core`), 852
- `_SUNAdaptControllerContent_ImExGus` (class in `sundials4py.core`), 852
- `_SUNAdaptControllerContent_Soderlind` (class in `sundials4py.core`), 852
- `_SUNLinearSolverContent_Band` (class in `sundials4py.core`), 852
- `_SUNLinearSolverContent_Dense` (class in `sundials4py.core`), 852
- `_SUNLinearSolverContent_PCG` (class in `sundials4py.core`), 852
- `_SUNLinearSolverContent_SPBCGS` (class in `sundials4py.core`), 852
- `_SUNLinearSolverContent_SPGMR` (class in `sundials4py.core`), 852
- `_SUNLinearSolverContent_SPGMR` (class in `sundials4py.core`), 852
- `_SUNLinearSolverContent_SPTFQMR` (class in `sundials4py.core`), 852
- `_SUNMatrixContent_Band` (class in `sundials4py.core`), 852
- `_SUNMatrixContent_Dense` (class in `sundials4py.core`), 853
- `_SUNMatrixContent_Sparse` (class in `sundials4py.core`), 853
- `_SUNNonlinearSolverContent_FixedPoint` (class in `sundials4py.core`), 853
- `_SUNNonlinearSolverContent_Newton` (class in `sundials4py.core`), 853
- `_braid_Vector_struct` (C struct), 273
- `_braid_Vector_struct.y` (C member), 273
- `_generic_N_Vector` (C struct), 433
- `_generic_N_Vector` (class in `sundials4py.core`), 853
- `_generic_N_Vector.content` (C member), 433
- `_generic_N_Vector.ops` (C member), 433
- `_generic_N_Vector.sunctx` (C member), 433
- `_generic_N_Vector_Ops` (C struct), 433
- `_generic_N_Vector_Ops` (class in `sundials4py.core`), 853
- `_generic_N_Vector_Ops.nvabs` (C member), 434
- `_generic_N_Vector_Ops.nvaddconst` (C member), 434
- `_generic_N_Vector_Ops.nvbufpack` (C member), 436
- `_generic_N_Vector_Ops.nvbufsize` (C member), 436
- `_generic_N_Vector_Ops.nvbufunpack` (C member), 436
- `_generic_N_Vector_Ops.nvclone` (C member), 434
- `_generic_N_Vector_Ops.nvcloneempty` (C member), 434
- `_generic_N_Vector_Ops.nvcompare` (C member), 435
- `_generic_N_Vector_Ops.nvconst` (C member), 434
- `_generic_N_Vector_Ops.nvconstrmask` (C member), 435
- `_generic_N_Vector_Ops.nvconstrmasklocal` (C member), 436
- `_generic_N_Vector_Ops.nvconstvectorarray` (C member), 435
- `_generic_N_Vector_Ops.nvdestroy` (C member), 434
- `_generic_N_Vector_Ops.nvdiv` (C member), 434
- `_generic_N_Vector_Ops.nvdotprod` (C member), 435
- `_generic_N_Vector_Ops.nvdotprodlocal` (C member), 436
- `_generic_N_Vector_Ops.nvdotprodmulti` (C member), 435
- `_generic_N_Vector_Ops.nvdotprodmultiallreduce` (C member), 436
- `_generic_N_Vector_Ops.nvdotprodmultilocal` (C member), 436
- `_generic_N_Vector_Ops.nvgetarraypointer` (C member), 434
- `_generic_N_Vector_Ops.nvgetcommunicator` (C member), 434
- `_generic_N_Vector_Ops.nvgetdevicearraypointer` (C member), 434
- `_generic_N_Vector_Ops.nvgetlength` (C member), 434
- `_generic_N_Vector_Ops.nvgetlocallength` (C member), 434

member), 434  
 \_generic\_N\_Vector\_Ops.nvgetvectorid (C member), 434  
 \_generic\_N\_Vector\_Ops.nvinv (C member), 434  
 \_generic\_N\_Vector\_Ops.nvinvtest (C member), 435  
 \_generic\_N\_Vector\_Ops.nvinvtestlocal (C member), 436  
 \_generic\_N\_Vector\_Ops.nvlnorm (C member), 435  
 \_generic\_N\_Vector\_Ops.nvlnormlocal (C member), 436  
 \_generic\_N\_Vector\_Ops.nvlinearcombination (C member), 435  
 \_generic\_N\_Vector\_Ops.nvlinearcombinationvectorarray (C member), 436  
 \_generic\_N\_Vector\_Ops.nvlinearsum (C member), 434  
 \_generic\_N\_Vector\_Ops.nvlinearsumvectorarray (C member), 435  
 \_generic\_N\_Vector\_Ops.nvmaxnorm (C member), 435  
 \_generic\_N\_Vector\_Ops.nvmaxnormlocal (C member), 436  
 \_generic\_N\_Vector\_Ops.nvmin (C member), 435  
 \_generic\_N\_Vector\_Ops.nvminlocal (C member), 436  
 \_generic\_N\_Vector\_Ops.nvminquotient (C member), 435  
 \_generic\_N\_Vector\_Ops.nvminquotientlocal (C member), 436  
 \_generic\_N\_Vector\_Ops.nvprint (C member), 436  
 \_generic\_N\_Vector\_Ops.nvprintfile (C member), 436  
 \_generic\_N\_Vector\_Ops.nvprod (C member), 434  
 \_generic\_N\_Vector\_Ops.nvscale (C member), 434  
 \_generic\_N\_Vector\_Ops.nvscaleaddmulti (C member), 435  
 \_generic\_N\_Vector\_Ops.nvscaleaddmultivectorarray (C member), 436  
 \_generic\_N\_Vector\_Ops.nvscalevectorarray (C member), 435  
 \_generic\_N\_Vector\_Ops.nvsetarraypointer (C member), 434  
 \_generic\_N\_Vector\_Ops.nvspace (C member), 434  
 \_generic\_N\_Vector\_Ops.nvw12norm (C member), 435  
 \_generic\_N\_Vector\_Ops.nvwrmsnorm (C member), 435  
 \_generic\_N\_Vector\_Ops.nvwrmsnormmask (C member), 435  
 \_generic\_N\_Vector\_Ops.nvwrmsnormmaskvectorarray (C member), 435  
 \_generic\_N\_Vector\_Ops.nvwrmsnormvectorarray (C member), 435  
 \_generic\_N\_Vector\_Ops.nvwsqrsumlocal (C member), 436  
 \_generic\_N\_Vector\_Ops.nvwsqrsummasklocal (C member), 436  
 \_generic\_SUNAdaptController (C struct), 663  
 \_generic\_SUNAdaptController (class in sundials4py.core), 853  
 \_generic\_SUNAdaptController.content (C member), 663  
 \_generic\_SUNAdaptController.ops (C member), 663  
 \_generic\_SUNAdaptController.sunctx (C member), 663  
 \_generic\_SUNAdaptController\_Ops (C struct), 664  
 \_generic\_SUNAdaptController\_Ops (class in sundials4py.core), 853  
 \_generic\_SUNAdaptController\_Ops.destroy (C member), 664  
 \_generic\_SUNAdaptController\_Ops.estimatestep (C member), 664  
 \_generic\_SUNAdaptController\_Ops.estimatestep\_tol (C member), 664  
 \_generic\_SUNAdaptController\_Ops.gettype (C member), 664  
 \_generic\_SUNAdaptController\_Ops.reset (C member), 664  
 \_generic\_SUNAdaptController\_Ops.setdefaults (C member), 664  
 \_generic\_SUNAdaptController\_Ops.seterrorbias (C member), 664  
 \_generic\_SUNAdaptController\_Ops.setoptions (C member), 664  
 \_generic\_SUNAdaptController\_Ops.space (C member), 664  
 \_generic\_SUNAdaptController\_Ops.updateh (C member), 664  
 \_generic\_SUNAdaptController\_Ops.updatemritol (C member), 664  
 \_generic\_SUNAdaptController\_Ops.write (C member), 664  
 \_generic\_SUNLinearSolver (C struct), 563  
 \_generic\_SUNLinearSolver (class in sundials4py.core), 853  
 \_generic\_SUNLinearSolver.content (C member), 563  
 \_generic\_SUNLinearSolver.ops (C member), 564  
 \_generic\_SUNLinearSolver.sunctx (C member), 564  
 \_generic\_SUNLinearSolver\_Ops (C struct), 564  
 \_generic\_SUNLinearSolver\_Ops (class in sundi-

`als4py.core)`, 853  
`_generic_SUNLinearSolver_Ops.free` (*C member*), 565  
`_generic_SUNLinearSolver_Ops.getid` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.gettype` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.initialize` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.lastflag` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.numiters` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.resid` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.resnorm` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.setatimes` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.setoptions` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.setpreconditioner` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.setscalingvectors` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.setup` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.setzeroguess` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.solve` (*C member*), 564  
`_generic_SUNLinearSolver_Ops.space` (*C member*), 564  
`_generic_SUNMatrix` (*C struct*), 511  
`_generic_SUNMatrix` (*class in sundials4py.core*), 853  
`_generic_SUNMatrix.content` (*C member*), 511  
`_generic_SUNMatrix.ops` (*C member*), 511  
`_generic_SUNMatrix.sunctx` (*C member*), 511  
`_generic_SUNMatrix_Ops` (*C struct*), 511  
`_generic_SUNMatrix_Ops` (*class in sundials4py.core*), 853  
`_generic_SUNMatrix_Ops.clone` (*C member*), 512  
`_generic_SUNMatrix_Ops.copy` (*C member*), 512  
`_generic_SUNMatrix_Ops.destroy` (*C member*), 512  
`_generic_SUNMatrix_Ops.getid` (*C member*), 512  
`_generic_SUNMatrix_Ops.mathermitiantransposevec` (*C member*), 512  
`_generic_SUNMatrix_Ops.matvec` (*C member*), 512  
`_generic_SUNMatrix_Ops.matvecsetup` (*C member*), 512  
`_generic_SUNMatrix_Ops.scaleadd` (*C member*), 512  
`_generic_SUNMatrix_Ops.scaleaddi` (*C member*), 512  
`_generic_SUNMatrix_Ops.space` (*C member*), 512  
`_generic_SUNMatrix_Ops.zero` (*C member*), 512  
`_generic_SUNNonlinearSolver` (*C struct*), 631  
`_generic_SUNNonlinearSolver` (*class in sundials4py.core*), 853  
`_generic_SUNNonlinearSolver.content` (*C member*), 631  
`_generic_SUNNonlinearSolver.ops` (*C member*), 631  
`_generic_SUNNonlinearSolver.sunctx` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops` (*C struct*), 631  
`_generic_SUNNonlinearSolver_Ops` (*class in sundials4py.core*), 853  
`_generic_SUNNonlinearSolver_Ops.free` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops.getcuriter` (*C member*), 632  
`_generic_SUNNonlinearSolver_Ops.getnumconvfails` (*C member*), 632  
`_generic_SUNNonlinearSolver_Ops.getnumiters` (*C member*), 632  
`_generic_SUNNonlinearSolver_Ops.gettype` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops.initialize` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops.setctestfn` (*C member*), 632  
`_generic_SUNNonlinearSolver_Ops.setlsetupfn` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops.setlsolvefn` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops.setmaxiters` (*C member*), 632  
`_generic_SUNNonlinearSolver_Ops.setsysfn` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops.setup` (*C member*), 631  
`_generic_SUNNonlinearSolver_Ops.solve` (*C member*), 631

## A

additive Runge--Kutta methods, 14

Adjoint Sensitivity Analysis user main program, 418

ARK\_ACCUMERROR\_AVG (*sundials4py.arkode.ARKAccumError attribute*), 878

ARK\_ACCUMERROR\_MAX (*sundials4py.arkode.ARKAccumError attribute*), 878

ARK_ACCUMERROR_NONE	( <i>sundials4py.arkode.ARKAccumError</i> attribute), 878	ARK_POSTSTEPFN_FAIL, 779	
ARK_ACCUMERROR_SUM	( <i>sundials4py.arkode.ARKAccumError</i> attribute), 878	ARK_PRERHS_FAIL, 779	
ARK_ADJ_CHECKPOINT_FAIL, 779		ARK_PRESTEPFN_FAIL, 779	
ARK_ADJ_RECOMPUTE_FAIL, 779		ARK_RELAX_BRENT, 775	
ARK_BAD_DKY, 779		ARK_RELAX_BRENT	( <i>sundials4py.arkode.ARKRelaxSolver</i> attribute), 883
ARK_BAD_K, 778		ARK_RELAX_FAIL, 779	
ARK_BAD_T, 778		ARK_RELAX_FUNC_FAIL, 779	
ARK_CONSTR_FAIL, 778		ARK_RELAX_JAC_FAIL, 779	
ARK_CONTEXT_ERR, 779		ARK_RELAX_MEM_FAIL, 779	
ARK_CONTROLLER_ERR, 779		ARK_RELAX_NEWTON, 775	
ARK_CONV_FAILURE, 778		ARK_RELAX_NEWTON	( <i>sundials4py.arkode.ARKRelaxSolver</i> attribute), 883
ARK_DEE_FAIL, 779		ARK_REPTD_RHSFUNC_ERR, 778	
ARK_DOMEIG_FAIL, 779		ARK_RHSFUNC_FAIL, 778	
ARK_ERR_FAILURE, 778		ARK_ROOT_RETURN, 778	
ARK_FIRST_RHSFUNC_ERR, 778		ARK_RTFUNC_FAIL, 778	
ARK_FULLRHS_END, 775		ARK_STEP_DIRECTION_ERR, 779	
ARK_FULLRHS_OTHER, 775		ARK_STEP_H0_FAIL, 779	
ARK_FULLRHS_START, 775		ARK_STEPPER_UNSUPPORTED, 779	
ARK_ILL_INPUT, 778		ARK_SUCCESS, 778	
ARK_INNERSTEP_ATTACH_ERR, 779		ARK_SUNADJSTEPPER_ERR, 779	
ARK_INNERSTEP_FAIL, 779		ARK_SUNSTEPPER_ERR, 779	
ARK_INNERTOOUTER_FAIL, 779		ARK_TOO_CLOSE, 779	
ARK_INTERP_FAIL, 779		ARK_TOO_MUCH_ACC, 778	
ARK_INTERP_HERMITE, 775		ARK_TOO_MUCH_WORK, 778	
ARK_INTERP_LAGRANGE, 775		ARK_TSTOP_RETURN, 778	
ARK_INTERP_MAX_DEGREE, 775		ARK_UNREC_RHSFUNC_ERR, 778	
ARK_INTERP_NONE, 775		ARK_UNRECOGNIZED_ERROR, 779	
ARK_INVALID_TABLE, 779		ARK_USER_PREDICT_FAIL, 779	
ARK_LFREE_FAIL, 778		ARK_VECTOROP_ERR, 779	
ARK_LINIT_FAIL, 778		ARK_WARNING, 778	
ARK_LSETUP_FAIL, 778		ARKAccumError ( <i>C enum</i> ), 108	
ARK_LSOLVE_FAIL, 778		ARKAccumError ( <i>class in sundials4py.arkode</i> ), 878	
ARK_MASSFREE_FAIL, 778		ARKAccumError.ARK_ACCUMERROR_AVG ( <i>C enumerator</i> ), 109	
ARK_MASSINIT_FAIL, 778		ARKAccumError.ARK_ACCUMERROR_MAX ( <i>C enumerator</i> ), 109	
ARK_MASSMULT_FAIL, 778		ARKAccumError.ARK_ACCUMERROR_NONE ( <i>C enumerator</i> ), 108	
ARK_MASSSETUP_FAIL, 778		ARKAccumError.ARK_ACCUMERROR_SUM ( <i>C enumerator</i> ), 109	
ARK_MASSSOLVE_FAIL, 778		ARKAdaptFn ( <i>C type</i> ), 173	
ARK_MAX_STAGE_LIMIT_FAIL, 779		ARKBandPrecGetNumRhsEvals ( <i>C function</i> ), 196	
ARK_MEM_FAIL, 778		ARKBandPrecGetWorkSpace ( <i>C function</i> ), 195	
ARK_MEM_NULL, 778		ARKBandPrecInit ( <i>C function</i> ), 195	
ARK_NLS_INIT_FAIL, 779		ARKBBDPrecGetNumGfnEvals ( <i>C function</i> ), 202	
ARK_NLS_OP_ERR, 779		ARKBBDPrecGetWorkSpace ( <i>C function</i> ), 201	
ARK_NLS_SETUP_FAIL, 779		ARKBBDPrecInit ( <i>C function</i> ), 200	
ARK_NLS_SETUP_RECVR, 779		ARKBBDPrecReInit ( <i>C function</i> ), 201	
ARK_NO_MALLOC, 778		ARKBraid_Access ( <i>C function</i> ), 283	
ARK_NORMAL, 775		ARKBraid_BraidInit ( <i>C function</i> ), 277	
ARK_ONE_STEP, 775			
ARK_OUTERTOINNER_FAIL, 779			
ARK_POSTPROCESS_STAGE_FAIL, 779			
ARK_POSTPROCESS_STEP_FAIL, 779			

- ARKBraid\_Create (C function), 277  
 ARKBraid\_Free (C function), 278  
 ARKBraid\_GetARKodeMem (C function), 280  
 ARKBraid\_GetARKStepMem (C function), 280  
 ARKBraid\_GetLastARKodeFlag (C function), 281  
 ARKBraid\_GetLastARKStepFlag (C function), 281  
 ARKBraid\_GetLastBraidFlag (C function), 281  
 ARKBraid\_GetSolution (C function), 281  
 ARKBraid\_GetUserData (C function), 280  
 ARKBraid\_GetVecTpl (C function), 280  
 ARKBraid\_Init (C function), 283  
 ARKBraid\_SetAccessFn (C function), 279  
 ARKBraid\_SetInitFn (C function), 278  
 ARKBraid\_SetSpatialNormFn (C function), 279  
 ARKBraid\_SetStepFn (C function), 278  
 ARKBraid\_Step (C function), 282  
 ARKBraid\_TakeStep (C function), 285  
 ARKCommFn (C type), 198  
 ARKDomEigFn (C type), 333  
 ARKEwtFn (C type), 171  
 ARKExpStabFn (C type), 174  
 ARKLocalFn (C type), 198  
 ARKLS\_ILL\_INPUT, 780  
 ARKLS\_JACFUNC\_RECVR, 780  
 ARKLS\_JACFUNC\_UNRECVR, 780  
 ARKLS\_LMEM\_NULL, 780  
 ARKLS\_MASSFUNC\_RECVR, 780  
 ARKLS\_MASSFUNC\_UNRECVR, 780  
 ARKLS\_MASSMEM\_NULL, 780  
 ARKLS\_MEM\_FAIL, 780  
 ARKLS\_MEM\_NULL, 780  
 ARKLS\_PMEM\_NULL, 780  
 ARKLS\_SUCCESS, 780  
 ARKLS\_SUNLS\_FAIL, 780  
 ARKLS\_SUNMAT\_FAIL, 780  
 ARKLSJacFn (C type), 175  
 ARKLSJacTimesSetupFn (C type), 178  
 ARKLSJacTimesVecFn (C type), 177  
 ARKLSLinSysFn (C type), 176  
 ARKLSMassFn (C type), 180  
 ARKLSMassPrecSetupFn (C type), 183  
 ARKLSMassPrecSolveFn (C type), 182  
 ARKLSMassTimesSetupFn (C type), 182  
 ARKLSMassTimesVecFn (C type), 181  
 ARKLSPrecSetupFn (C type), 179  
 ARKLSPrecSolveFn (C type), 179  
 ARKODE\_ARK2\_DIRK\_3\_1\_2 (C enumerator), 805  
 ARKODE\_ARK2\_DIRK\_3\_1\_2 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 878  
 ARKODE\_ARK2\_ERK\_3\_1\_2 (C enumerator), 786  
 ARKODE\_ARK2\_ERK\_3\_1\_2 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 879  
 ARKODE\_ARK324L2SA\_DIRK\_4\_2\_3 (C enumerator), 811  
 ARKODE\_ARK324L2SA\_DIRK\_4\_2\_3 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ARK324L2SA\_ERK\_4\_2\_3 (C enumerator), 787  
 ARKODE\_ARK324L2SA\_ERK\_4\_2\_3 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_ARK436L2SA\_DIRK\_6\_3\_4 (C enumerator), 815  
 ARKODE\_ARK436L2SA\_DIRK\_6\_3\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ARK436L2SA\_ERK\_6\_3\_4 (C enumerator), 791  
 ARKODE\_ARK436L2SA\_ERK\_6\_3\_4 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_ARK437L2SA\_DIRK\_7\_3\_4 (C enumerator), 815  
 ARKODE\_ARK437L2SA\_DIRK\_7\_3\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ARK437L2SA\_ERK\_7\_3\_4 (C enumerator), 792  
 ARKODE\_ARK437L2SA\_ERK\_7\_3\_4 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_ARK548L2SA\_DIRK\_8\_4\_5 (C enumerator), 819  
 ARKODE\_ARK548L2SA\_DIRK\_8\_4\_5 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ARK548L2SA\_ERK\_8\_4\_5 (C enumerator), 797  
 ARKODE\_ARK548L2SA\_ERK\_8\_4\_5 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_ARK548L2SAb\_DIRK\_8\_4\_5 (C enumerator), 821  
 ARKODE\_ARK548L2SAb\_DIRK\_8\_4\_5 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ARK548L2SAb\_ERK\_8\_4\_5 (C enumerator), 797  
 ARKODE\_ARK548L2SAb\_ERK\_8\_4\_5 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_BACKWARD\_EULER\_1\_1 (C enumerator), 805  
 ARKODE\_BACKWARD\_EULER\_1\_1 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_BILLINGTON\_3\_3\_2 (C enumerator), 806  
 ARKODE\_BILLINGTON\_3\_3\_2 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_BOGACKI\_SHAMPINE\_4\_2\_3 (C enumerator), 787  
 ARKODE\_BOGACKI\_SHAMPINE\_4\_2\_3 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880



ARKODE\_CASH\_5\_2\_4 (C enumerator), 813  
 ARKODE\_CASH\_5\_2\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_CASH\_5\_3\_4 (C enumerator), 813  
 ARKODE\_CASH\_5\_3\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_CASH\_KARP\_6\_4\_5 (C enumerator), 793  
 ARKODE\_CASH\_KARP\_6\_4\_5 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_DIRK\_NONE (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_DIRKTableID (C enum), 803  
 ARKODE\_DIRKTableID (class in sundials4py.arkode), 878  
 ARKODE\_DORMAND\_PRINCE\_7\_4\_5 (C enumerator), 796  
 ARKODE\_DORMAND\_PRINCE\_7\_4\_5 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_ERK\_NONE (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_ERKTableID (C enum), 782  
 ARKODE\_ERKTableID (class in sundials4py.arkode), 879  
 ARKODE\_ESDIRK324L2SA\_4\_2\_3 (C enumerator), 809  
 ARKODE\_ESDIRK324L2SA\_4\_2\_3 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ESDIRK325L2SA\_5\_2\_3 (C enumerator), 808  
 ARKODE\_ESDIRK325L2SA\_5\_2\_3 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ESDIRK32I5L2SA\_5\_2\_3 (C enumerator), 809  
 ARKODE\_ESDIRK32I5L2SA\_5\_2\_3 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ESDIRK436L2SA\_6\_3\_4 (C enumerator), 811  
 ARKODE\_ESDIRK436L2SA\_6\_3\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ESDIRK437L2SA\_7\_3\_4 (C enumerator), 819  
 ARKODE\_ESDIRK437L2SA\_7\_3\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ESDIRK43I6L2SA\_6\_3\_4 (C enumerator), 817  
 ARKODE\_ESDIRK43I6L2SA\_6\_3\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ESDIRK547L2SA2\_7\_4\_5 (C enumerator), 819  
 ARKODE\_ESDIRK547L2SA2\_7\_4\_5 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_ESDIRK547L2SA\_7\_4\_5 (C enumerator), 821  
 ARKODE\_ESDIRK547L2SA\_7\_4\_5 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_EXPLICIT\_MIDPOINT\_EULER\_2\_1\_2 (C enumerator), 785  
 ARKODE\_EXPLICIT\_MIDPOINT\_EULER\_2\_1\_2 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_FEHLBERG\_13\_7\_8 (C enumerator), 801  
 ARKODE\_FEHLBERG\_13\_7\_8 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_FEHLBERG\_6\_4\_5 (C enumerator), 795  
 ARKODE\_FEHLBERG\_6\_4\_5 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_FORWARD\_EULER\_1\_1 (C enumerator), 783  
 ARKODE\_FORWARD\_EULER\_1\_1 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_HEUN\_EULER\_2\_1\_2 (C enumerator), 784  
 ARKODE\_HEUN\_EULER\_2\_1\_2 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_IMEX\_MRI\_GARK3a, 382  
 ARKODE\_IMEX\_MRI\_GARK3a (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMEX\_MRI\_GARK3b, 382  
 ARKODE\_IMEX\_MRI\_GARK3b (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMEX\_MRI\_GARK4, 382  
 ARKODE\_IMEX\_MRI\_GARK4 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMEX\_MRI\_GARK\_EULER, 382  
 ARKODE\_IMEX\_MRI\_GARK\_EULER (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMEX\_MRI\_GARK\_MIDPOINT, 382  
 ARKODE\_IMEX\_MRI\_GARK\_MIDPOINT (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMEX\_MRI\_GARK\_TRAPEZOIDAL, 382  
 ARKODE\_IMEX\_MRI\_GARK\_TRAPEZOIDAL (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMEX\_MRI\_SR21, 382  
 ARKODE\_IMEX\_MRI\_SR21 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMEX\_MRI\_SR32, 382  
 ARKODE\_IMEX\_MRI\_SR32 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881

ARKODE\_IMEX\_MRI\_SR43, 382  
 ARKODE\_IMEX\_MRI\_SR43 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_IMPLICIT\_MIDPOINT\_1\_2 (C enumerator), 806  
 ARKODE\_IMPLICIT\_MIDPOINT\_1\_2 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_IMPLICIT\_TRAPEZOIDAL\_2\_2 (C enumerator), 806  
 ARKODE\_IMPLICIT\_TRAPEZOIDAL\_2\_2 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_KNOTH\_WOLKE\_3\_3 (C enumerator), 788  
 ARKODE\_KNOTH\_WOLKE\_3\_3 (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_KVAERNO\_4\_2\_3 (C enumerator), 809  
 ARKODE\_KVAERNO\_4\_2\_3 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_KVAERNO\_5\_3\_4 (C enumerator), 815  
 ARKODE\_KVAERNO\_5\_3\_4 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_KVAERNO\_7\_4\_5 (C enumerator), 819  
 ARKODE\_KVAERNO\_7\_4\_5 (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_LSRK\_RKC\_2, 777  
 ARKODE\_LSRK\_RKC\_2 (sundials4py.arkode.ARKODE\_LSRKMethodType attribute), 880  
 ARKODE\_LSRK\_RKL\_2, 777  
 ARKODE\_LSRK\_RKL\_2 (sundials4py.arkode.ARKODE\_LSRKMethodType attribute), 881  
 ARKODE\_LSRK\_SSP\_10\_4, 777  
 ARKODE\_LSRK\_SSP\_10\_4 (sundials4py.arkode.ARKODE\_LSRKMethodType attribute), 881  
 ARKODE\_LSRK\_SSP\_S\_2, 777  
 ARKODE\_LSRK\_SSP\_S\_2 (sundials4py.arkode.ARKODE\_LSRKMethodType attribute), 881  
 ARKODE\_LSRK\_SSP\_S\_3, 777  
 ARKODE\_LSRK\_SSP\_S\_3 (sundials4py.arkode.ARKODE\_LSRKMethodType attribute), 881  
 ARKODE\_LSRKMethodType (C enum), 325  
 ARKODE\_LSRKMethodType (class in sundials4py.arkode), 880  
 ARKODE\_LSRKMethodType.ARKODE\_LSRK\_RKC\_2 (C enumerator), 325  
 ARKODE\_LSRKMethodType.ARKODE\_LSRK\_RKL\_2 (C enumerator), 325  
 ARKODE\_LSRKMethodType.ARKODE\_LSRK\_SSP\_10\_4 (C enumerator), 325  
 ARKODE\_LSRKMethodType.ARKODE\_LSRK\_SSP\_S\_2 (C enumerator), 325  
 ARKODE\_LSRKMethodType.ARKODE\_LSRK\_SSP\_S\_3 (C enumerator), 325  
 ARKODE\_MAX\_DIRK\_NUM (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_MAX\_ERK\_NUM (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_MAX\_MRI\_NUM (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MAX\_SPLITTING\_NUM (sundials4py.arkode.ARKODE\_SplittingCoefficientsID attribute), 882  
 ARKODE\_MAX\_SPRK\_NUM (sundials4py.arkode.ARKODE\_SPRKMethodID attribute), 882  
 ARKODE\_MERK21, 381  
 ARKODE\_MERK21 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MERK32, 381  
 ARKODE\_MERK32 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MERK43, 381  
 ARKODE\_MERK43 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MERK54, 20, 381  
 ARKODE\_MERK54 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MIN\_DIRK\_NUM (sundials4py.arkode.ARKODE\_DIRKTableID attribute), 879  
 ARKODE\_MIN\_ERK\_NUM (sundials4py.arkode.ARKODE\_ERKTableID attribute), 880  
 ARKODE\_MIN\_MRI\_NUM (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MIN\_SPLITTING\_NUM (sundials4py.arkode.ARKODE\_SplittingCoefficientsID attribute), 882  
 ARKODE\_MIN\_SPRK\_NUM (sundials4py.arkode.ARKODE\_SPRKMethodID attribute), 882  
 ARKODE\_MIS\_KW3, 381  
 ARKODE\_MIS\_KW3 (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MRI\_GARK\_BACKWARD\_EULER, 381  
 ARKODE\_MRI\_GARK\_BACKWARD\_EULER (sundials4py.arkode.ARKODE\_MRITableID attribute), 881  
 ARKODE\_MRI\_GARK\_ERK22a, 381  
 ARKODE\_MRI\_GARK\_ERK22a (sundials4py.arkode.ARKODE\_MRITableID attribute), 881

tribute), 881		ARKODE_RALSTON_EULER_2_1_2 (sundi-
ARKODE_MRI_GARK_ERK22b, 381		als4py.arkode.ARKODE_ERKTableID at-
ARKODE_MRI_GARK_ERK22b	(sundi-	tribute), 880
als4py.arkode.ARKODE_MRITableID at-		ARKODE_SAYFY_ABURUB_6_3_4 (C enumerator), 793
tribute), 881		ARKODE_SAYFY_ABURUB_6_3_4 (sundi-
ARKODE_MRI_GARK_ERK33a, 381		als4py.arkode.ARKODE_ERKTableID at-
ARKODE_MRI_GARK_ERK33a	(sundi-	tribute), 880
als4py.arkode.ARKODE_MRITableID at-		ARKODE_SDIRK_2_1_2 (C enumerator), 805
tribute), 881		ARKODE_SDIRK_2_1_2 (sundials4py.arkode.ARKODE_-
ARKODE_MRI_GARK_ERK45a, 381		DIRKTableID attribute), 879
ARKODE_MRI_GARK_ERK45a	(sundi-	ARKODE_SDIRK_5_3_4 (C enumerator), 814
als4py.arkode.ARKODE_MRITableID at-		ARKODE_SDIRK_5_3_4 (sundials4py.arkode.ARKODE_-
tribute), 881		DIRKTableID attribute), 879
ARKODE_MRI_GARK_ESDIRK34a, 381		ARKODE_SHU_OSHER_3_2_3 (C enumerator), 788
ARKODE_MRI_GARK_ESDIRK34a	(sundi-	ARKODE_SHU_OSHER_3_2_3 (sundi-
als4py.arkode.ARKODE_MRITableID at-		als4py.arkode.ARKODE_ERKTableID at-
tribute), 881		tribute), 880
ARKODE_MRI_GARK_ESDIRK46a, 381		ARKODE_SOFRONIOU_SPALETTA_5_3_4 (C enumerator),
ARKODE_MRI_GARK_ESDIRK46a	(sundi-	790
als4py.arkode.ARKODE_MRITableID at-		ARKODE_SOFRONIOU_SPALETTA_5_3_4 (sundi-
tribute), 881		als4py.arkode.ARKODE_ERKTableID at-
ARKODE_MRI_GARK_FORWARD_EULER, 381		tribute), 880
ARKODE_MRI_GARK_FORWARD_EULER	(sundi-	ARKODE_SPLITTING_BEST_2_2_2 (sundi-
als4py.arkode.ARKODE_MRITableID at-		als4py.arkode.ARKODE_SplittingCoeffi-
tribute), 881		cientsID attribute), 882
ARKODE_MRI_GARK_IMPLICIT_MIDPOINT, 381		ARKODE_SPLITTING_LIE_TROTTER_1_1_2 (sundi-
ARKODE_MRI_GARK_IMPLICIT_MIDPOINT	(sundi-	als4py.arkode.ARKODE_SplittingCoeffi-
als4py.arkode.ARKODE_MRITableID at-		cientsID attribute), 882
tribute), 881		ARKODE_SPLITTING_NONE (sundi-
ARKODE_MRI_GARK_IRK21a, 381		als4py.arkode.ARKODE_SplittingCoeffi-
ARKODE_MRI_GARK_IRK21a	(sundi-	cientsID attribute), 882
als4py.arkode.ARKODE_MRITableID at-		ARKODE_SPLITTING_RUTH_3_3_2 (sundi-
tribute), 882		als4py.arkode.ARKODE_SplittingCoeffi-
ARKODE_MRI_GARK_RALSTON2, 381		cientsID attribute), 882
ARKODE_MRI_GARK_RALSTON2	(sundi-	ARKODE_SPLITTING_STRANG_2_2_2 (sundi-
als4py.arkode.ARKODE_MRITableID at-		als4py.arkode.ARKODE_SplittingCoeffi-
tribute), 882		cientsID attribute), 882
ARKODE_MRI_GARK_RALSTON3, 381		ARKODE_SPLITTING_SUZUKI_3_3_2 (sundi-
ARKODE_MRI_GARK_RALSTON3	(sundi-	als4py.arkode.ARKODE_SplittingCoeffi-
als4py.arkode.ARKODE_MRITableID at-		cientsID attribute), 882
tribute), 882		ARKODE_SPLITTING_YOSHIDA_4_4_2 (sundi-
ARKODE_MRI_NONE (sundials4py.arkode.ARKODE_MRI-		als4py.arkode.ARKODE_SplittingCoeffi-
TableID attribute), 882		cientsID attribute), 882
ARKODE_MRITableID (C type), 380		ARKODE_SPLITTING_YOSHIDA_8_6_2 (sundi-
ARKODE_MRITableID (class in sundials4py.arkode), 881		als4py.arkode.ARKODE_SplittingCoeffi-
ARKODE_QESDIRK436L2SA_6_3_4 (C enumerator), 817		cientsID attribute), 882
ARKODE_QESDIRK436L2SA_6_3_4	(sundi-	ARKODE_SplittingCoefficientsID (C enum), 402
als4py.arkode.ARKODE_DIRKTableID at-		ARKODE_SplittingCoefficientsID (class in sundi-
tribute), 879		als4py.arkode), 882
ARKODE_RALSTON_3_1_2 (C enumerator), 783		ARKODE_SPRK_CANDY_ROZMUS_4_4 (C enumerator), 825
ARKODE_RALSTON_3_1_2	(sundi-	ARKODE_SPRK_CANDY_ROZMUS_4_4 (sundi-
als4py.arkode.ARKODE_ERKTableID at-		als4py.arkode.ARKODE_SPRKMethodID
tribute), 880		attribute), 882
ARKODE_RALSTON_EULER_2_1_2 (C enumerator), 785		ARKODE_SPRK_EULER_1_1 (C enumerator), 824



ARKODE_SPRK_EULER_1_1	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_TSITOURAS_7_4_5 (C enumerator), 793	
ARKODE_SPRK_LEAPFROG_2_2 (C enumerator), 824		ARKODE_TSITOURAS_7_4_5	( <i>sundials4py.arkode.ARKODE_ERKTableID</i> attribute), 880
ARKODE_SPRK_LEAPFROG_2_2	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_VERNER_10_6_7 (C enumerator), 800	
ARKODE_SPRK_MCLACHLAN_2_2 (C enumerator), 824		ARKODE_VERNER_10_6_7	( <i>sundials4py.arkode.ARKODE_ERKTableID</i> attribute), 880
ARKODE_SPRK_MCLACHLAN_2_2	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_VERNER_13_7_8 (C enumerator), 800	
ARKODE_SPRK_MCLACHLAN_3_3 (C enumerator), 824		ARKODE_VERNER_13_7_8	( <i>sundials4py.arkode.ARKODE_ERKTableID</i> attribute), 880
ARKODE_SPRK_MCLACHLAN_3_3	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_VERNER_16_8_9 (C enumerator), 802	
ARKODE_SPRK_MCLACHLAN_4_4 (C enumerator), 824		ARKODE_VERNER_16_8_9	( <i>sundials4py.arkode.ARKODE_ERKTableID</i> attribute), 880
ARKODE_SPRK_MCLACHLAN_4_4	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_VERNER_8_5_6 (C enumerator), 798	
ARKODE_SPRK_MCLACHLAN_5_6 (C enumerator), 825		ARKODE_VERNER_8_5_6	( <i>sundials4py.arkode.ARKODE_ERKTableID</i> attribute), 880
ARKODE_SPRK_MCLACHLAN_5_6	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_VERNER_9_5_6 (C enumerator), 798	
ARKODE_SPRK_NONE	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_VERNER_9_5_6	( <i>sundials4py.arkode.ARKODE_ERKTableID</i> attribute), 880
ARKODE_SPRK_PSEUDO_LEAPFROG_2_2 (C enumerator), 824		ARKODE_ZONNEVELD_5_3_4 (C enumerator), 790	
ARKODE_SPRK_PSEUDO_LEAPFROG_2_2	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKODE_ZONNEVELD_5_3_4	( <i>sundials4py.arkode.ARKODE_ERKTableID</i> attribute), 880
ARKODE_SPRK_RUTH_3_3 (C enumerator), 824		ARKodeButcherTable (C type), 423	
ARKODE_SPRK_RUTH_3_3	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKodeButcherTable_Alloc (C function), 425	
ARKODE_SPRK_SOFRONIOU_10_36 (C enumerator), 825		ARKodeButcherTable_CheckARKOrder (C function), 428	
ARKODE_SPRK_SOFRONIOU_10_36	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKodeButcherTable_CheckOrder (C function), 427	
ARKODE_SPRK_SUZUKI_UMENO_8_16 (C enumerator), 825		ARKodeButcherTable_Copy (C function), 426	
ARKODE_SPRK_SUZUKI_UMENO_8_16	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKodeButcherTable_Create (C function), 426	
ARKODE_SPRK_YOSHIDA_6_8 (C enumerator), 825		ARKodeButcherTable_DIRKIDToName (C function), 425	
ARKODE_SPRK_YOSHIDA_6_8	( <i>sundials4py.arkode.ARKODE_SPRKMethodID</i> attribute), 882	ARKodeButcherTable_ERKIDToName (C function), 424	
ARKODE_SPRKMethodID (C enum), 823		ARKodeButcherTable_Free (C function), 427	
ARKODE_SPRKMethodID (class in <i>sundials4py.arkode</i> ), 882		ARKodeButcherTable_IsStifflyAccurate (C function), 427	
ARKODE_TRBDF2_3_3_2 (C enumerator), 808		ARKodeButcherTable_LoadDIRK (C function), 425	
ARKODE_TRBDF2_3_3_2	( <i>sundials4py.arkode.ARKODE_DIRKTableID</i> attribute), 879	ARKodeButcherTable_LoadDIRKByName (C function), 425	
		ARKodeButcherTable_LoadERK (C function), 424	
		ARKodeButcherTable_LoadERKByName (C function), 424	
		ARKodeButcherTable_Space (C function), 426	
		ARKodeButcherTable_Write (C function), 427	
		ARKodeButcherTableMem (C struct), 423	
		ARKodeButcherTableMem (class in <i>sundials4py.arkode</i> ), 883	
		ARKodeButcherTableMem.A (C member), 423	
		ARKodeButcherTableMem.b (C member), 423	

- ARKodeButcherTableMem.c (C member), 423
- ARKodeButcherTableMem.d (C member), 423
- ARKodeButcherTableMem.p (C member), 423
- ARKodeButcherTableMem.q (C member), 423
- ARKodeButcherTableMem.stages (C member), 423
- ARKodeClearStopTime (C function), 97
- ARKodeComputeState (C function), 635
- ARKodeCreateMRISStepInnerStepper (C function), 168
- ARKodeCreateSUNStepper (C function), 169
- ARKodeEvolve (C function), 85
- ARKodeFree (C function), 76
- ARKodeGetAccumulatedError (C function), 147
- ARKodeGetActualInitStep (C function), 139
- ARKodeGetCurrentGamma (C function), 141
- ARKodeGetCurrentMassMatrix (C function), 634
- ARKodeGetCurrentState (C function), 141
- ARKodeGetCurrentStep (C function), 139
- ARKodeGetCurrentTime (C function), 140
- ARKodeGetDky (C function), 136
- ARKodeGetErrWeights (C function), 142
- ARKodeGetEstLocalErrors (C function), 146
- ARKodeGetJac (C function), 152
- ARKodeGetJacNumSteps (C function), 153
- ARKodeGetJacTime (C function), 153
- ARKodeGetLastLinFlag (C function), 158
- ARKodeGetLastMassFlag (C function), 164
- ARKodeGetLastState (C function), 140
- ARKodeGetLastStep (C function), 139
- ARKodeGetLastTime (C function), 140
- ARKodeGetLinReturnFlagName (C function), 159
- ARKodeGetLinWorkSpace (C function), 153
- ARKodeGetMassWorkSpace (C function), 159
- ARKodeGetNonlinearSystemData (C function), 634
- ARKodeGetNonlinSolvStats (C function), 150
- ARKodeGetNumAccSteps (C function), 144
- ARKodeGetNumConstrFails (C function), 146
- ARKodeGetNumErrTestFails (C function), 145
- ARKodeGetNumExpSteps (C function), 144
- ARKodeGetNumGEvals (C function), 151
- ARKodeGetNumJacEvals (C function), 154
- ARKodeGetNumJtimesEvals (C function), 157
- ARKodeGetNumJTSetupEvals (C function), 156
- ARKodeGetNumLinConvFails (C function), 156
- ARKodeGetNumLinIters (C function), 155
- ARKodeGetNumLinRhsEvals (C function), 157
- ARKodeGetNumLinSolvSetups (C function), 148
- ARKodeGetNumMassConvFails (C function), 163
- ARKodeGetNumMassIters (C function), 163
- ARKodeGetNumMassMult (C function), 161
- ARKodeGetNumMassMultSetups (C function), 160
- ARKodeGetNumMassPrecEvals (C function), 162
- ARKodeGetNumMassPrecSolves (C function), 162
- ARKodeGetNumMassSetups (C function), 160
- ARKodeGetNumMassSolves (C function), 161
- ARKodeGetNumMTSetups (C function), 163
- ARKodeGetNumNonlinSolvConvFails (C function), 149
- ARKodeGetNumNonlinSolvIters (C function), 149
- ARKodeGetNumPrecEvals (C function), 154
- ARKodeGetNumPrecSolves (C function), 155
- ARKodeGetNumRelaxBoundFails (C function), 192
- ARKodeGetNumRelaxFails (C function), 192
- ARKodeGetNumRelaxFnEvals (C function), 192
- ARKodeGetNumRelaxJacEvals (C function), 192
- ARKodeGetNumRelaxSolveFails (C function), 193
- ARKodeGetNumRelaxSolveIters (C function), 193
- ARKodeGetNumRhsEvals (C function), 145
- ARKodeGetNumStepAttempts (C function), 144
- ARKodeGetNumSteps (C function), 138
- ARKodeGetNumStepSolveFails (C function), 146
- ARKodeGetResWeights (C function), 142
- ARKodeGetReturnFlagName (C function), 143
- ARKodeGetRootInfo (C function), 150
- ARKodeGetStageIndex (C function), 147
- ARKodeGetStepDirection (C function), 140
- ARKodeGetStepStats (C function), 142
- ARKodeGetTolScaleFactor (C function), 141
- ARKodeGetUserData (C function), 147
- ARKodeGetWorkSpace (C function), 138
- ARKodeInit (C function), 165
- ARKodePrintAllStats (C function), 143
- ARKodeReset (C function), 166
- ARKodeResetAccumulatedError (C function), 109
- ARKodeResFtolerance (C function), 79
- ARKodeResize (C function), 167
- ARKodeResStolerance (C function), 78
- ARKodeResVtolerance (C function), 79
- ARKodeRootInit (C function), 85
- ARKodeSetAccumulatedErrorType (C function), 109
- ARKodeSetAdaptController (C function), 101
- ARKodeSetAdaptControllerByName (C function), 101
- ARKodeSetAdaptivityAdjustment (C function), 102
- ARKodeSetAdjointCheckpointIndex (C function), 100
- ARKodeSetAdjointCheckpointScheme (C function), 99
- ARKodeSetAutonomous (C function), 111
- ARKodeSetCFLFraction (C function), 102
- ARKodeSetConstraints (C function), 98
- ARKodeSetDeduceImplicitRhs (C function), 116
- ARKodeSetDefaults (C function), 89
- ARKodeSetDeltaGammaMax (C function), 118
- ARKodeSetEpsLin (C function), 127
- ARKodeSetErrorBias (C function), 103
- ARKodeSetFixedStep (C function), 92
- ARKodeSetFixedStepBounds (C function), 104
- ARKodeSetInitStep (C function), 93

- ARKodeSetInterpolantDegree (*C function*), 91  
 ARKodeSetInterpolantType (*C function*), 90  
 ARKodeSetInterpolateStopTime (*C function*), 96  
 ARKodeSetJacEvalFrequency (*C function*), 119  
 ARKodeSetJacFn (*C function*), 120  
 ARKodeSetJacTimes (*C function*), 123  
 ARKodeSetJacTimesRhsFn (*C function*), 124  
 ARKodeSetLinear (*C function*), 110  
 ARKodeSetLinearSolutionScaling (*C function*), 122  
 ARKodeSetLinearSolver (*C function*), 81  
 ARKodeSetLinSysFn (*C function*), 121  
 ARKodeSetLSetupFrequency (*C function*), 118  
 ARKodeSetLSNormFactor (*C function*), 128  
 ARKodeSetMassEpsLin (*C function*), 128  
 ARKodeSetMassFn (*C function*), 121  
 ARKodeSetMassLinearSolver (*C function*), 83  
 ARKodeSetMassLSNormFactor (*C function*), 129  
 ARKodeSetMassPreconditioner (*C function*), 126  
 ARKodeSetMassTimes (*C function*), 124  
 ARKodeSetMaxCFailGrowth (*C function*), 104  
 ARKodeSetMaxConvFails (*C function*), 116  
 ARKodeSetMaxEFailGrowth (*C function*), 105  
 ARKodeSetMaxErrTestFails (*C function*), 97  
 ARKodeSetMaxFirstGrowth (*C function*), 105  
 ARKodeSetMaxGrowth (*C function*), 106  
 ARKodeSetMaxHnilWarns (*C function*), 94  
 ARKodeSetMaxNonlinIters (*C function*), 114  
 ARKodeSetMaxNumConstrFails (*C function*), 99  
 ARKodeSetMaxNumSteps (*C function*), 94  
 ARKodeSetMaxStep (*C function*), 95  
 ARKodeSetMinReduction (*C function*), 106  
 ARKodeSetMinStep (*C function*), 95  
 ARKodeSetNlsRhsFn (*C function*), 113  
 ARKodeSetNoInactiveRootWarn (*C function*), 130  
 ARKodeSetNonlinConvCoef (*C function*), 114  
 ARKodeSetNonlinCRDown (*C function*), 115  
 ARKodeSetNonlinear (*C function*), 111  
 ARKodeSetNonlinearSolver (*C function*), 84  
 ARKodeSetNonlinRDiv (*C function*), 115  
 ARKodeSetOptions (*C function*), 88  
 ARKodeSetOrder (*C function*), 90  
 ARKodeSetPostprocessStageFn (*C function*), 135  
 ARKodeSetPostprocessStepFn (*C function*), 134  
 ARKodeSetPostStepFn (*C function*), 133  
 ARKodeSetPreconditioner (*C function*), 126  
 ARKodeSetPredictorMethod (*C function*), 112  
 ARKodeSetPreRhsFn (*C function*), 134  
 ARKodeSetPreStepFn (*C function*), 133  
 ARKodeSetRelaxEtaFail (*C function*), 189  
 ARKodeSetRelaxFn (*C function*), 188  
 ARKodeSetRelaxLowerBound (*C function*), 189  
 ARKodeSetRelaxMaxFails (*C function*), 190  
 ARKodeSetRelaxMaxIters (*C function*), 190  
 ARKodeSetRelaxResTol (*C function*), 191  
 ARKodeSetRelaxSolver (*C function*), 190  
 ARKodeSetRelaxTol (*C function*), 191  
 ARKodeSetRelaxUpperBound (*C function*), 189  
 ARKodeSetRootDirection (*C function*), 130  
 ARKodeSetSafetyFactor (*C function*), 107  
 ARKodeSetSmallNumEFails (*C function*), 107  
 ARKodeSetStabilityFn (*C function*), 108  
 ARKodeSetStagePredictFn (*C function*), 113  
 ARKodeSetStepDirection (*C function*), 93  
 ARKodeSetStopTime (*C function*), 96  
 ARKodeSetUseCompensatedSums (*C function*), 100  
 ARKodeSetUserData (*C function*), 97  
 ARKodeSPRKTable (*C type*), 429  
 ARKodeSPRKTable\_Alloc (*C function*), 430  
 ARKodeSPRKTable\_Copy (*C function*), 430  
 ARKodeSPRKTable\_Create (*C function*), 430  
 ARKodeSPRKTable\_Free (*C function*), 431  
 ARKodeSPRKTable\_Load (*C function*), 430  
 ARKodeSPRKTable\_LoadByName (*C function*), 430  
 ARKodeSPRKTable\_Space (*C function*), 431  
 ARKodeSPRKTable\_ToButcher (*C function*), 431  
 ARKodeSPRKTable\_Write (*C function*), 431  
 ARKodeSPRKTableMem (*C type*), 429  
 ARKodeSPRKTableMem (*class in sundials4py.arkode*), 883  
 ARKodeSPRKTableMem.a (*C member*), 429  
 ARKodeSPRKTableMem.ahat (*C member*), 429  
 ARKodeSPRKTableMem.q (*C member*), 429  
 ARKodeSPRKTableMem.stages (*C member*), 429  
 ARKodeSStolerances (*C function*), 77  
 ARKodeSVtolerances (*C function*), 77  
 ARKodeView (*class in sundials4py.arkode*), 883  
 ARKodeWftolerances (*C function*), 78  
 ARKodeWriteParameters (*C function*), 165  
 ARKPostProcessFn (*C type*), 187  
 ARKPostStepFn (*C type*), 186  
 ARKPreRhsFn (*C type*), 187  
 ARKPreStepFn (*C type*), 186  
 ARKRelaxFn (*C type*), 185  
 ARKRelaxJacFn (*C type*), 185  
 ARKRelaxSolver (*C enum*), 780  
 ARKRelaxSolver (*class in sundials4py.arkode*), 883  
 ARKRelaxSolver.ARK\_RELAX\_BRENT (*C enumerator*), 780  
 ARKRelaxSolver.ARK\_RELAX\_NEWTON (*C enumerator*), 780  
 ARKRhsFn (*C type*), 171  
 ARKRootFn (*C type*), 175  
 ARKRwtFn (*C type*), 172  
 ARKStagePredictFn (*C type*), 174  
 ARKSTEP\_DEFAULT\_ARK\_ETABLE\_2, 776  
 ARKSTEP\_DEFAULT\_ARK\_ETABLE\_3, 776  
 ARKSTEP\_DEFAULT\_ARK\_ETABLE\_4, 776  
 ARKSTEP\_DEFAULT\_ARK\_ETABLE\_5, 777



- ARKSTEP\_DEFAULT\_ARK\_ITABLE\_2, 776
- ARKSTEP\_DEFAULT\_ARK\_ITABLE\_3, 776
- ARKSTEP\_DEFAULT\_ARK\_ITABLE\_4, 776
- ARKSTEP\_DEFAULT\_ARK\_ITABLE\_5, 777
- ARKSTEP\_DEFAULT\_DIRK\_1, 776
- ARKSTEP\_DEFAULT\_DIRK\_2, 776
- ARKSTEP\_DEFAULT\_DIRK\_3, 776
- ARKSTEP\_DEFAULT\_DIRK\_4, 776
- ARKSTEP\_DEFAULT\_DIRK\_5, 776
- ARKSTEP\_DEFAULT\_ERK\_1, 775
- ARKSTEP\_DEFAULT\_ERK\_2, 775
- ARKSTEP\_DEFAULT\_ERK\_3, 775
- ARKSTEP\_DEFAULT\_ERK\_4, 776
- ARKSTEP\_DEFAULT\_ERK\_5, 776
- ARKSTEP\_DEFAULT\_ERK\_6, 776
- ARKSTEP\_DEFAULT\_ERK\_7, 776
- ARKSTEP\_DEFAULT\_ERK\_8, 776
- ARKSTEP\_DEFAULT\_ERK\_9, 776
- ARKStepClearStopTime (C function), 214
- ARKStepComputeState (C function), 637
- ARKStepCreate (C function), 203
- ARKStepCreateAdjointStepper (C function), 420
- ARKStepCreateMRISStepInnerStepper (C function), 263
- ARKStepEvolve (C function), 208
- ARKStepFree (C function), 203
- ARKStepGetActualInitStep (C function), 242
- ARKStepGetCurrentButcherTables (C function), 247
- ARKStepGetCurrentGamma (C function), 243
- ARKStepGetCurrentMassMatrix (C function), 636
- ARKStepGetCurrentState (C function), 243
- ARKStepGetCurrentStep (C function), 242
- ARKStepGetCurrentTime (C function), 243
- ARKStepGetDky (C function), 241
- ARKStepGetErrWeights (C function), 244
- ARKStepGetEstLocalErrors (C function), 247
- ARKStepGetJac (C function), 251
- ARKStepGetJacNumSteps (C function), 252
- ARKStepGetJacTime (C function), 251
- ARKStepGetLastLinFlag (C function), 255
- ARKStepGetLastMassFlag (C function), 259
- ARKStepGetLastStep (C function), 242
- ARKStepGetLinReturnFlagName (C function), 256
- ARKStepGetLinWorkSpace (C function), 252
- ARKStepGetMassWorkSpace (C function), 256
- ARKStepGetNonlinearSystemData (C function), 636
- ARKStepGetNonlinSolvStats (C function), 250
- ARKStepGetNumAccSteps (C function), 246
- ARKStepGetNumConstrFails (C function), 248
- ARKStepGetNumErrTestFails (C function), 246
- ARKStepGetNumExpSteps (C function), 245
- ARKStepGetNumGEvals (C function), 251
- ARKStepGetNumJacEvals (C function), 252
- ARKStepGetNumJtimesEvals (C function), 254
- ARKStepGetNumJTSetupEvals (C function), 254
- ARKStepGetNumLinConvFails (C function), 254
- ARKStepGetNumLinIters (C function), 253
- ARKStepGetNumLinRhsEvals (C function), 255
- ARKStepGetNumLinSolvSetups (C function), 249
- ARKStepGetNumMassConvFails (C function), 259
- ARKStepGetNumMassIters (C function), 258
- ARKStepGetNumMassMult (C function), 257
- ARKStepGetNumMassMultSetups (C function), 257
- ARKStepGetNumMassPrecEvals (C function), 258
- ARKStepGetNumMassPrecSolves (C function), 258
- ARKStepGetNumMassSetups (C function), 256
- ARKStepGetNumMassSolves (C function), 257
- ARKStepGetNumMTSetups (C function), 259
- ARKStepGetNumNonlinSolvConvFails (C function), 250
- ARKStepGetNumNonlinSolvIters (C function), 249
- ARKStepGetNumPrecEvals (C function), 253
- ARKStepGetNumPrecSolves (C function), 253
- ARKStepGetNumRelaxBoundFails (C function), 269
- ARKStepGetNumRelaxFails (C function), 269
- ARKStepGetNumRelaxFnEvals (C function), 268
- ARKStepGetNumRelaxJacEvals (C function), 268
- ARKStepGetNumRelaxSolveFails (C function), 269
- ARKStepGetNumRelaxSolveIters (C function), 270
- ARKStepGetNumRhsEvals (C function), 246
- ARKStepGetNumStepAttempts (C function), 246
- ARKStepGetNumSteps (C function), 241
- ARKStepGetNumStepSolveFails (C function), 247
- ARKStepGetResWeights (C function), 244
- ARKStepGetReturnFlagName (C function), 245
- ARKStepGetRootInfo (C function), 250
- ARKStepGetStepStats (C function), 244
- ARKStepGetTimestepperStats (C function), 248
- ARKStepGetTolScaleFactor (C function), 243
- ARKStepGetUserData (C function), 249
- ARKStepGetWorkSpace (C function), 241
- ARKStepPrintAllStats (C function), 245
- ARKStepReInit (C function), 261
- ARKStepReset (C function), 262
- ARKStepResFtolerance (C function), 205
- ARKStepResize (C function), 262
- ARKStepResStolerance (C function), 204
- ARKStepResVtolerance (C function), 205
- ARKStepRootInit (C function), 207
- ARKStepSetAdaptController (C function), 222
- ARKStepSetAdaptivityAdjustment (C function), 224
- ARKStepSetAdaptivityFn (C function), 223
- ARKStepSetAdaptivityMethod (C function), 223
- ARKStepSetCFLFraction (C function), 224
- ARKStepSetConstraints (C function), 218
- ARKStepSetDeduceImplicitRhs (C function), 232
- ARKStepSetDefaults (C function), 210
- ARKStepSetDeltaGammaMax (C function), 232

- ARKStepSetDenseOrder (*C function*), 211
- ARKStepSetDiagnostics (*C function*), 211
- ARKStepSetEpsLin (*C function*), 238
- ARKStepSetErrorBias (*C function*), 224
- ARKStepSetExplicit (*C function*), 220
- ARKStepSetFixedStep (*C function*), 211
- ARKStepSetFixedStepBounds (*C function*), 225
- ARKStepSetImEx (*C function*), 219
- ARKStepSetImplicit (*C function*), 220
- ARKStepSetInitStep (*C function*), 212
- ARKStepSetInterpolantDegree (*C function*), 210
- ARKStepSetInterpolantType (*C function*), 210
- ARKStepSetInterpolateStopTime (*C function*), 214
- ARKStepSetJacEvalFrequency (*C function*), 233
- ARKStepSetJacFn (*C function*), 234
- ARKStepSetJacTimes (*C function*), 235
- ARKStepSetJacTimesRhsFn (*C function*), 236
- ARKStepSetLinear (*C function*), 228
- ARKStepSetLinearSolutionScaling (*C function*), 235
- ARKStepSetLinearSolver (*C function*), 206
- ARKStepSetLinSysFn (*C function*), 234
- ARKStepSetLSetupFrequency (*C function*), 233
- ARKStepSetLSNormFactor (*C function*), 239
- ARKStepSetMassEpsLin (*C function*), 238
- ARKStepSetMassFn (*C function*), 234
- ARKStepSetMassLinearSolver (*C function*), 206
- ARKStepSetMassLSNormFactor (*C function*), 239
- ARKStepSetMassPreconditioner (*C function*), 237
- ARKStepSetMassTimes (*C function*), 236
- ARKStepSetMaxCFailGrowth (*C function*), 225
- ARKStepSetMaxConvFails (*C function*), 231
- ARKStepSetMaxEFailGrowth (*C function*), 225
- ARKStepSetMaxErrTestFails (*C function*), 215
- ARKStepSetMaxFirstGrowth (*C function*), 226
- ARKStepSetMaxGrowth (*C function*), 226
- ARKStepSetMaxHnilWarns (*C function*), 212
- ARKStepSetMaxNonlinIters (*C function*), 230
- ARKStepSetMaxNumConstrFails (*C function*), 218
- ARKStepSetMaxNumSteps (*C function*), 212
- ARKStepSetMaxStep (*C function*), 213
- ARKStepSetMinReduction (*C function*), 226
- ARKStepSetMinStep (*C function*), 213
- ARKStepSetNlsRhsFn (*C function*), 230
- ARKStepSetNoInactiveRootWarn (*C function*), 240
- ARKStepSetNonlinConvCoef (*C function*), 230
- ARKStepSetNonlinCRDown (*C function*), 231
- ARKStepSetNonlinear (*C function*), 228
- ARKStepSetNonlinearSolver (*C function*), 207
- ARKStepSetNonlinRDiv (*C function*), 231
- ARKStepSetOptimalParams (*C function*), 215
- ARKStepSetOrder (*C function*), 219
- ARKStepSetPreconditioner (*C function*), 237
- ARKStepSetPredictorMethod (*C function*), 229
- ARKStepSetRelaxEtaFail (*C function*), 265
- ARKStepSetRelaxFn (*C function*), 264
- ARKStepSetRelaxLowerBound (*C function*), 265
- ARKStepSetRelaxMaxFails (*C function*), 266
- ARKStepSetRelaxMaxIters (*C function*), 266
- ARKStepSetRelaxResTol (*C function*), 267
- ARKStepSetRelaxSolver (*C function*), 267
- ARKStepSetRelaxTol (*C function*), 268
- ARKStepSetRelaxUpperBound (*C function*), 266
- ARKStepSetRootDirection (*C function*), 240
- ARKStepSetSafetyFactor (*C function*), 227
- ARKStepSetSmallNumEFails (*C function*), 227
- ARKStepSetStabilityFn (*C function*), 228
- ARKStepSetStagePredictFn (*C function*), 229
- ARKStepSetStopTime (*C function*), 213
- ARKStepSetTableName (*C function*), 222
- ARKStepSetTableNum (*C function*), 221
- ARKStepSetTables (*C function*), 220
- ARKStepSetUserData (*C function*), 214
- ARKStepSStolerances (*C function*), 203
- ARKStepSVtolerances (*C function*), 204
- ARKStepWftolerances (*C function*), 204
- ARKStepWriteButcher (*C function*), 260
- ARKStepWriteParameters (*C function*), 260
- ARKVecResizeFn (*C type*), 183

## C

### CMake options

- adiak\_DIR, 732
- AMDGPU\_TARGETS, 736
- BLA\_VENDOR, 740
- BLAS\_LIBRARIES, 740
- BLAS\_LINKER\_FLAGS, 740
- BUILD\_SHARED\_LIBS, 728
- BUILD\_STATIC\_LIBS, 728
- CALIPER\_DIR, 733
- CMAKE\_BUILD\_TYPE, 725
- CMAKE\_C\_COMPILER, 725
- CMAKE\_C\_EXTENSIONS, 726
- CMAKE\_C\_FLAGS, 725
- CMAKE\_C\_FLAGS\_DEBUG, 725
- CMAKE\_C\_FLAGS\_MINSIZEREL, 726
- CMAKE\_C\_FLAGS\_RELEASE, 725
- CMAKE\_C\_FLAGS\_RELWITHDEBINFO, 725
- CMAKE\_C\_STANDARD, 726
- CMAKE\_CONFIGURATION\_TYPES, 725
- CMAKE\_CUDA\_ARCHITECTURES, 734
- CMAKE\_CXX\_COMPILER, 726
- CMAKE\_CXX\_EXTENSIONS, 726
- CMAKE\_CXX\_FLAGS, 726
- CMAKE\_CXX\_FLAGS\_DEBUG, 726
- CMAKE\_CXX\_FLAGS\_MINSIZEREL, 726
- CMAKE\_CXX\_FLAGS\_RELEASE, 726
- CMAKE\_CXX\_FLAGS\_RELWITHDEBINFO, 726

CMAKE\_CXX\_STANDARD, 726  
CMAKE\_Fortran\_COMPILER, 727  
CMAKE\_Fortran\_FLAGS, 727  
CMAKE\_Fortran\_FLAGS\_DEBUG, 727  
CMAKE\_Fortran\_FLAGS\_MINSIZEREL, 727  
CMAKE\_Fortran\_FLAGS\_RELEASE, 727  
CMAKE\_Fortran\_FLAGS\_RELWITHDEBINFO, 727  
CMAKE\_INSTALL\_LIBDIR, 727  
CMAKE\_INSTALL\_PREFIX, 727  
CUDA\_TOOLKIT\_ROOT\_DIR, 734  
Ginkgo\_DIR, 735  
HYPRE\_DIR, 737  
KLU\_INCLUDE\_DIR, 737  
KLU\_LIBRARY\_DIR, 737  
KLU\_ROOT, 737  
Kokkos\_DIR, 738  
KokkosKernels\_DIR, 739  
LAPACK\_LIBRARIES, 740  
LAPACK\_LINKER\_FLAGS, 740  
LAPACK\_ROOT, 740  
MAGMA\_DIR, 741  
MPI\_C\_COMPILER, 742  
MPI\_CXX\_COMPILER, 742  
MPI\_Fortran\_COMPILER, 743  
MPIEXEC\_EXECUTABLE, 743  
MPIEXEC\_POSTFLAGS, 743  
MPIEXEC\_PREFLAGS, 743  
ONEMKL\_DIR, 744  
PETSC\_DIR, 745  
PETSC\_INCLUDES, 746  
PETSC\_LIBRARIES, 746  
RAJA\_DIR, 747  
SUNDIALS\_ENABLE\_ADIK, 732  
SUNDIALS\_ENABLE\_ADIK\_CHECKS, 733  
SUNDIALS\_ENABLE\_ARKODE, 729  
SUNDIALS\_ENABLE\_C\_EXAMPLES, 730  
SUNDIALS\_ENABLE\_CALIPER, 733  
SUNDIALS\_ENABLE\_CALIPER\_CHECKS, 733  
SUNDIALS\_ENABLE\_CUDA, 734  
SUNDIALS\_ENABLE\_CUDA\_EXAMPLES, 730  
SUNDIALS\_ENABLE\_CVODE, 729  
SUNDIALS\_ENABLE\_CVODES, 729  
SUNDIALS\_ENABLE\_CXX\_EXAMPLES, 730  
SUNDIALS\_ENABLE\_ERROR\_CHECKS, 731  
SUNDIALS\_ENABLE\_EXAMPLES\_INSTALL, 730  
SUNDIALS\_ENABLE\_EXTERNAL\_ADDONS, 753  
SUNDIALS\_ENABLE\_FORTRAN, 730  
SUNDIALS\_ENABLE\_FORTRAN\_EXAMPLES, 730  
SUNDIALS\_ENABLE\_GINKGO, 735  
SUNDIALS\_ENABLE\_GINKGO\_CHECKS, 735  
SUNDIALS\_ENABLE\_HIP, 736  
SUNDIALS\_ENABLE\_HYPRE, 736  
SUNDIALS\_ENABLE\_HYPRE\_CHECKS, 737  
SUNDIALS\_ENABLE\_IDA, 729  
SUNDIALS\_ENABLE\_IDAS, 729  
SUNDIALS\_ENABLE\_KINSOL, 729  
SUNDIALS\_ENABLE\_KLU, 737  
SUNDIALS\_ENABLE\_KLU\_CHECKS, 737  
SUNDIALS\_ENABLE\_KOKKOS, 738  
SUNDIALS\_ENABLE\_KOKKOS\_CHECKS, 738  
SUNDIALS\_ENABLE\_KOKKOS\_KERNELS, 739  
SUNDIALS\_ENABLE\_KOKKOS\_KERNELS\_CHECKS, 739  
SUNDIALS\_ENABLE\_LAPACK, 740  
SUNDIALS\_ENABLE\_LAPACK\_CHECKS, 741  
SUNDIALS\_ENABLE\_MAGMA, 741  
SUNDIALS\_ENABLE\_MAGMA\_CHECKS, 742  
SUNDIALS\_ENABLE\_MONITORING, 732  
SUNDIALS\_ENABLE\_MPI, 742  
SUNDIALS\_ENABLE\_ONEMKL, 743  
SUNDIALS\_ENABLE\_ONEMKL\_CHECKS, 744  
SUNDIALS\_ENABLE\_OPENMP, 744  
SUNDIALS\_ENABLE\_OPENMP\_DEVICE, 745  
SUNDIALS\_ENABLE\_OPENMP\_DEVICE\_CHECKS, 745  
SUNDIALS\_ENABLE\_PACKAGE\_FUSED\_KERNELS, 732  
SUNDIALS\_ENABLE\_PETSC, 745  
SUNDIALS\_ENABLE\_PETSC\_CHECKS, 746  
SUNDIALS\_ENABLE\_PROFILING, 732  
SUNDIALS\_ENABLE\_PTHREAD, 746  
SUNDIALS\_ENABLE\_RAJA, 747  
SUNDIALS\_ENABLE\_SUPERLUDIST, 747  
SUNDIALS\_ENABLE\_SUPERLUDIST\_CHECKS, 749  
SUNDIALS\_ENABLE\_SUPERLUMT, 749  
SUNDIALS\_ENABLE\_SUPERLUMT\_CHECKS, 750  
SUNDIALS\_ENABLE\_SYCL, 750  
SUNDIALS\_ENABLE\_TRILINOS, 751  
SUNDIALS\_ENABLE\_XBRAID, 752  
SUNDIALS\_ENABLE\_XBRAID\_CHECKS, 752  
SUNDIALS\_EXAMPLES\_INSTALL\_PATH, 730  
SUNDIALS\_GINKGO\_BACKENDS, 735  
SUNDIALS\_INDEX\_SIZE, 728  
SUNDIALS\_INDEX\_TYPE, 728  
SUNDIALS\_INSTALL\_CMAKEDIR, 728  
SUNDIALS\_LAPACK\_CASE, 740  
SUNDIALS\_LAPACK\_UNDESCORES, 741  
SUNDIALS\_LOGGING\_LEVEL, 731  
SUNDIALS\_MAGMA\_BACKENDS, 741  
SUNDIALS\_MATH\_LIBRARY, 729  
SUNDIALS\_ONEMKL\_USE\_GETRF\_LOOP, 744  
SUNDIALS\_ONEMKL\_USE\_GETRS\_LOOP, 744  
SUNDIALS\_PRECISION, 729  
SUNDIALS\_RAJA\_BACKENDS, 747  
SUNDIALS\_SYCL\_2020\_UNSUPPORTED, 751  
SUPERLUDIST\_DIR, 748  
SUPERLUDIST\_INCLUDE\_DIR, 748  
SUPERLUDIST\_INCLUDE\_DIRS, 748  
SUPERLUDIST\_LIBRARIES, 748

- SUPERLUDIST\_LIBRARY\_DIR, 748  
 SUPERLUDIST\_OpenMP, 748  
 SUPERLUMT\_INCLUDE\_DIR, 750  
 SUPERLUMT\_LIBRARIES, 750  
 SUPERLUMT\_LIBRARY\_DIR, 750  
 SUPERLUMT\_THREAD\_TYPE, 750  
 Trilinos\_DIR, 751  
 USE\_XSDK\_DEFAULTS, 752  
 XBRAID\_DIR, 752  
 XBRAID\_INCLUDES, 752  
 XBRAID\_LIBRARIES, 752  
 CopyFromDevice (C++ function), 493  
 CopyToDevice (C++ function), 493
- ## D
- DenseLinearSolver (C++ class), 619  
 DenseLinearSolver::~DenseLinearSolver (C++ function), 619  
 DenseLinearSolver::DenseLinearSolver (C++ function), 619  
 DenseLinearSolver::get (C++ function), 620  
 DenseLinearSolver::operator SUNLinearSolver (C++ function), 620  
 DenseLinearSolver::operator= (C++ function), 619  
 DenseMatrix (C++ class), 550  
 DenseMatrix::~DenseMatrix (C++ function), 552  
 DenseMatrix::BlockCols (C++ function), 552  
 DenseMatrix::BlockRows (C++ function), 552  
 DenseMatrix::Blocks (C++ function), 552  
 DenseMatrix::Cols (C++ function), 552  
 DenseMatrix::DenseMatrix (C++ function), 550, 551  
 DenseMatrix::exec\_space (C++ type), 550  
 DenseMatrix::ExecSpace (C++ function), 552  
 DenseMatrix::get (C++ function), 552  
 DenseMatrix::member\_type (C++ type), 550  
 DenseMatrix::memory\_space (C++ type), 550  
 DenseMatrix::operator SUNMatrix (C++ function), 552  
 DenseMatrix::operator= (C++ function), 551  
 DenseMatrix::range\_policy (C++ type), 550  
 DenseMatrix::Rows (C++ function), 552  
 DenseMatrix::size\_type (C++ type), 550  
 DenseMatrix::team\_policy (C++ type), 550  
 DenseMatrix::View (C++ function), 552  
 DenseMatrix::view\_type (C++ type), 550  
 diagonally-implicit Runge--Kutta methods, 15
- ## E
- ERKSTEP\_DEFAULT\_1, 776  
 ERKSTEP\_DEFAULT\_2, 776  
 ERKSTEP\_DEFAULT\_3, 776  
 ERKSTEP\_DEFAULT\_4, 776  
 ERKSTEP\_DEFAULT\_5, 776  
 ERKSTEP\_DEFAULT\_6, 776  
 ERKSTEP\_DEFAULT\_7, 776  
 ERKSTEP\_DEFAULT\_8, 776  
 ERKSTEP\_DEFAULT\_9, 776  
 ERKStepClearStopTime (C function), 294  
 ERKStepCreate (C function), 286  
 ERKStepCreateAdjointStepper (C function), 420  
 ERKStepEvolve (C function), 288  
 ERKStepFree (C function), 286  
 ERKStepGetActualInitStep (C function), 305  
 ERKStepGetCurrentButcherTable (C function), 309  
 ERKStepGetCurrentStep (C function), 306  
 ERKStepGetCurrentTime (C function), 306  
 ERKStepGetDky (C function), 304  
 ERKStepGetErrWeights (C function), 306  
 ERKStepGetEstLocalErrors (C function), 310  
 ERKStepGetLastStep (C function), 305  
 ERKStepGetNumAccSteps (C function), 308  
 ERKStepGetNumConstrFails (C function), 311  
 ERKStepGetNumErrTestFails (C function), 309  
 ERKStepGetNumExpSteps (C function), 308  
 ERKStepGetNumGEvals (C function), 312  
 ERKStepGetNumRelaxBoundFails (C function), 320  
 ERKStepGetNumRelaxFails (C function), 319  
 ERKStepGetNumRelaxFnEvals (C function), 319  
 ERKStepGetNumRelaxJacEvals (C function), 319  
 ERKStepGetNumRelaxSolveFails (C function), 320  
 ERKStepGetNumRelaxSolveIters (C function), 320  
 ERKStepGetNumRhsEvals (C function), 309  
 ERKStepGetNumStepAttempts (C function), 308  
 ERKStepGetNumSteps (C function), 305  
 ERKStepGetReturnFlagName (C function), 308  
 ERKStepGetRootInfo (C function), 311  
 ERKStepGetStepStats (C function), 307  
 ERKStepGetTimestepperStats (C function), 310  
 ERKStepGetTolScaleFactor (C function), 306  
 ERKStepGetUserData (C function), 311  
 ERKStepGetWorkspace (C function), 305  
 ERKStepPrintAllStats (C function), 307  
 ERKStepReInit (C function), 313  
 ERKStepReset (C function), 313  
 ERKStepResize (C function), 314  
 ERKStepRootInit (C function), 287  
 ERKStepSetAdaptController (C function), 298  
 ERKStepSetAdaptivityAdjustment (C function), 299  
 ERKStepSetAdaptivityFn (C function), 298  
 ERKStepSetAdaptivityMethod (C function), 299  
 ERKStepSetCFLFraction (C function), 300  
 ERKStepSetConstraints (C function), 295  
 ERKStepSetDefaults (C function), 289  
 ERKStepSetDenseOrder (C function), 290  
 ERKStepSetDiagnostics (C function), 290  
 ERKStepSetErrorBias (C function), 300  
 ERKStepSetFixedStep (C function), 291  
 ERKStepSetFixedStepBounds (C function), 300



ERKStepSetInitStep (*C function*), 291  
 ERKStepSetInterpolantDegree (*C function*), 290  
 ERKStepSetInterpolantType (*C function*), 289  
 ERKStepSetInterpolateStopTime (*C function*), 294  
 ERKStepSetMaxEFailGrowth (*C function*), 301  
 ERKStepSetMaxErrTestFails (*C function*), 294  
 ERKStepSetMaxFirstGrowth (*C function*), 301  
 ERKStepSetMaxGrowth (*C function*), 301  
 ERKStepSetMaxHnilWarns (*C function*), 292  
 ERKStepSetMaxNumConstrFails (*C function*), 295  
 ERKStepSetMaxNumSteps (*C function*), 292  
 ERKStepSetMaxStep (*C function*), 292  
 ERKStepSetMinReduction (*C function*), 302  
 ERKStepSetMinStep (*C function*), 293  
 ERKStepSetNoInactiveRootWarn (*C function*), 303  
 ERKStepSetOrder (*C function*), 296  
 ERKStepSetRelaxEtaFail (*C function*), 315  
 ERKStepSetRelaxFn (*C function*), 315  
 ERKStepSetRelaxLowerBound (*C function*), 316  
 ERKStepSetRelaxMaxFails (*C function*), 316  
 ERKStepSetRelaxMaxIters (*C function*), 317  
 ERKStepSetRelaxResTol (*C function*), 318  
 ERKStepSetRelaxSolver (*C function*), 317  
 ERKStepSetRelaxTol (*C function*), 318  
 ERKStepSetRelaxUpperBound (*C function*), 316  
 ERKStepSetRootDirection (*C function*), 303  
 ERKStepSetSafetyFactor (*C function*), 302  
 ERKStepSetSmallNumEFails (*C function*), 302  
 ERKStepSetStabilityFn (*C function*), 303  
 ERKStepSetStopTime (*C function*), 293  
 ERKStepSetTable (*C function*), 296  
 ERKStepSetTableName (*C function*), 297  
 ERKStepSetTableNum (*C function*), 297  
 ERKStepSetUserData (*C function*), 294  
 ERKStepSStolerances (*C function*), 286  
 ERKStepSVtolerances (*C function*), 287  
 ERKStepWFTolerances (*C function*), 287  
 ERKStepWriteButcher (*C function*), 312  
 ERKStepWriteParameters (*C function*), 312  
 error weight vector, 22  
 explicit Runge--Kutta methods, 15

## F

FILE (*class in sundials4py.core*), 848  
 fixed point iteration, 31  
 ForcingStepCreate (*C function*), 321  
 ForcingStepGetNumEvolves (*C function*), 322  
 ForcingStepReInit (*C function*), 323

## G

get (*sundials4py.arkode.ARKodeView attribute*), 883  
 get\_history() (*in module logs*), 59  
 GetDenseMat (*C++ function*), 552  
 GetVec (*C++ function*), 493

## I

inexact Newton iteration, 32

## L

linear solver setup, 32  
 log\_file\_to\_list() (*in module logs*), 58  
 LSRKStepCreateSSP (*C function*), 324  
 LSRKStepCreateSTS (*C function*), 324  
 LSRKStepGetMaxNumStages (*C function*), 330  
 LSRKStepGetNumDomEigEstIters (*C function*), 331  
 LSRKStepGetNumDomEigEstRhsEvals (*C function*), 330  
 LSRKStepGetNumDomEigUpdates (*C function*), 330  
 LSRKStepReInitSSP (*C function*), 332  
 LSRKStepReInitSTS (*C function*), 331  
 LSRKStepSetDomEigEstimator (*C function*), 326  
 LSRKStepSetDomEigFn (*C function*), 326  
 LSRKStepSetDomEigFrequency (*C function*), 327  
 LSRKStepSetDomEigSafetyFactor (*C function*), 328  
 LSRKStepSetMaxNumStages (*C function*), 327  
 LSRKStepSetNumDomEigEstInitPreprocessIters (*C function*), 328  
 LSRKStepSetNumDomEigEstPreprocessIters (*C function*), 329  
 LSRKStepSetNumSSPStages (*C function*), 329  
 LSRKStepSetSSPMethod (*C function*), 325  
 LSRKStepSetSSPMethodByName (*C function*), 326  
 LSRKStepSetSTSMethod (*C function*), 324  
 LSRKStepSetSTSMethodByName (*C function*), 325

## M

Matrix (*C++ class*), 546  
 Matrix::~Matrix (*C++ function*), 546  
 Matrix::get (*C++ function*), 547  
 Matrix::GkoExec (*C++ function*), 547  
 Matrix::GkoMtx (*C++ function*), 547  
 Matrix::GkoSize (*C++ function*), 547  
 Matrix::Matrix (*C++ function*), 546  
 Matrix::operator SUNMatrix (*C++ function*), 547  
 Matrix::operator= (*C++ function*), 546  
 modified Newton iteration, 31  
 module  
     sundials4py.arkode, 878  
     sundials4py.core, 848  
 MRISTep user main program, 333  
 MRISTEP\_DEFAULT\_EXPL\_1, 777  
 MRISTEP\_DEFAULT\_EXPL\_2, 777  
 MRISTEP\_DEFAULT\_EXPL\_2\_AD, 777  
 MRISTEP\_DEFAULT\_EXPL\_3, 777  
 MRISTEP\_DEFAULT\_EXPL\_3\_AD, 777  
 MRISTEP\_DEFAULT\_EXPL\_4, 777  
 MRISTEP\_DEFAULT\_EXPL\_4\_AD, 777  
 MRISTEP\_DEFAULT\_EXPL\_5\_AD, 777



MRISTEP\_DEFAULT\_IMEX\_SD\_1, 777  
 MRISTEP\_DEFAULT\_IMEX\_SD\_2, 777  
 MRISTEP\_DEFAULT\_IMEX\_SD\_2\_AD, 778  
 MRISTEP\_DEFAULT\_IMEX\_SD\_3, 777  
 MRISTEP\_DEFAULT\_IMEX\_SD\_3\_AD, 778  
 MRISTEP\_DEFAULT\_IMEX\_SD\_4, 778  
 MRISTEP\_DEFAULT\_IMEX\_SD\_4\_AD, 778  
 MRISTEP\_DEFAULT\_IMPL\_SD\_1, 777  
 MRISTEP\_DEFAULT\_IMPL\_SD\_2, 777  
 MRISTEP\_DEFAULT\_IMPL\_SD\_3, 777  
 MRISTEP\_DEFAULT\_IMPL\_SD\_4, 777  
 MRISTEP\_EXPLICIT, 777  
 MRISTEP\_EXPLICIT (*sundials4py.arkode.MRISTEP\_-METHOD\_TYPE* attribute), 883  
 MRISTEP\_IMEX, 777  
 MRISTEP\_IMEX (*sundials4py.arkode.MRISTEP\_-METHOD\_TYPE* attribute), 883  
 MRISTEP\_IMPLICIT, 777  
 MRISTEP\_IMPLICIT (*sundials4py.arkode.MRISTEP\_-METHOD\_TYPE* attribute), 883  
 MRISTEP\_MERK (*sundials4py.arkode.MRISTEP\_-METHOD\_TYPE* attribute), 883  
 MRISTEP\_METHOD\_TYPE (*C* enum), 375  
 MRISTEP\_METHOD\_TYPE (*class in sundials4py.arkode*), 883  
 MRISTEP\_METHOD\_TYPE.MRISTEP\_EXPLICIT (*C enumerator*), 375  
 MRISTEP\_METHOD\_TYPE.MRISTEP\_IMEX (*C enumerator*), 375  
 MRISTEP\_METHOD\_TYPE.MRISTEP\_IMPLICIT (*C enumerator*), 375  
 MRISTEP\_METHOD\_TYPE.MRISTEP\_MERK (*C enumerator*), 376  
 MRISTEP\_METHOD\_TYPE.MRISTEP\_SR (*C enumerator*), 376  
 MRISTEP\_SR (*sundials4py.arkode.MRISTEP\_-METHOD\_TYPE* attribute), 883  
 MRISTepClearStopTime (*C function*), 345  
 MRISTepComputeState (*C function*), 638  
 MRISTepCoupling (*C type*), 376  
 MRISTepCoupling\_Alloc (*C function*), 377  
 MRISTepCoupling\_Copy (*C function*), 379  
 MRISTepCoupling\_Create (*C function*), 378  
 MRISTepCoupling\_Free (*C function*), 380  
 MRISTepCoupling\_LoadTable (*C function*), 377  
 MRISTepCoupling\_LoadTableByName (*C function*), 377  
 MRISTepCoupling\_MISToMRI (*C function*), 379  
 MRISTepCoupling\_Space (*C function*), 380  
 MRISTepCoupling\_Write (*C function*), 380  
 MRISTepCouplingMem (*C struct*), 376  
 MRISTepCouplingMem (*class in sundials4py.arkode*), 883  
 MRISTepCouplingMem.c (*C member*), 376  
 MRISTepCouplingMem.G (*C member*), 376  
 MRISTepCouplingMem.group (*C member*), 376  
 MRISTepCouplingMem.ngroup (*C member*), 376  
 MRISTepCouplingMem.nmat (*C member*), 376  
 MRISTepCouplingMem.p (*C member*), 376  
 MRISTepCouplingMem.q (*C member*), 376  
 MRISTepCouplingMem.stages (*C member*), 376  
 MRISTepCouplingMem.type (*C member*), 376  
 MRISTepCouplingMem.W (*C member*), 376  
 MRISTepCreate (*C function*), 335  
 MRISTepEvolve (*C function*), 340  
 MRISTepFree (*C function*), 336  
 MRISTepGetCurrentCoupling (*C function*), 362  
 MRISTepGetCurrentGamma (*C function*), 360  
 MRISTepGetCurrentState (*C function*), 360  
 MRISTepGetCurrentTime (*C function*), 360  
 MRISTepGetDky (*C function*), 358  
 MRISTepGetErrWeights (*C function*), 361  
 MRISTepGetJac (*C function*), 366  
 MRISTepGetJacNumSteps (*C function*), 366  
 MRISTepGetJacTime (*C function*), 366  
 MRISTepGetLastInnerStepFlag (*C function*), 363  
 MRISTepGetLastLinFlag (*C function*), 370  
 MRISTepGetLastStep (*C function*), 359  
 MRISTepGetLinReturnFlagName (*C function*), 371  
 MRISTepGetLinWorkSpace (*C function*), 367  
 MRISTepGetNonlinearSystemData (*C function*), 637  
 MRISTepGetNonlinSolvStats (*C function*), 365  
 MRISTepGetNumGEvals (*C function*), 365  
 MRISTepGetNumInnerStepperFails (*C function*), 359  
 MRISTepGetNumJacEvals (*C function*), 367  
 MRISTepGetNumJtimesEvals (*C function*), 370  
 MRISTepGetNumJTSetupEvals (*C function*), 369  
 MRISTepGetNumLinConvFails (*C function*), 369  
 MRISTepGetNumLinIters (*C function*), 368  
 MRISTepGetNumLinRhsEvals (*C function*), 370  
 MRISTepGetNumLinSolvSetups (*C function*), 363  
 MRISTepGetNumNonlinSolvConvFails (*C function*), 364  
 MRISTepGetNumNonlinSolvIters (*C function*), 364  
 MRISTepGetNumPrecEvals (*C function*), 368  
 MRISTepGetNumPrecSolves (*C function*), 368  
 MRISTepGetNumRhsEvals (*C function*), 362  
 MRISTepGetNumSteps (*C function*), 359  
 MRISTepGetNumStepSolveFails (*C function*), 362  
 MRISTepGetReturnFlagName (*C function*), 362  
 MRISTepGetRootInfo (*C function*), 365  
 MRISTepGetTolScaleFactor (*C function*), 360  
 MRISTepGetUserData (*C function*), 363  
 MRISTepGetWorkSpace (*C function*), 359  
 MRISTepInnerEvolveFn (*C type*), 389  
 MRISTepInnerFullRhsFn (*C type*), 389  
 MRISTepInnerGetAccumulatedError (*C type*), 390  
 MRISTepInnerResetAccumulatedError (*C type*), 390

MRISetInnerResetFn (*C type*), 390  
MRISetInnerSetRTol (*C type*), 391  
MRISetInnerStepper (*C type*), 382  
MRISetInnerStepper\_AddForcing (*C function*), 387  
MRISetInnerStepper\_Create (*C function*), 383  
MRISetInnerStepper\_CreateFromSUNStepper (*C function*), 383  
MRISetInnerStepper\_Free (*C function*), 384  
MRISetInnerStepper\_GetContent (*C function*), 385  
MRISetInnerStepper\_GetForcingData (*C function*), 388  
MRISetInnerStepper\_SetAccumulatedError-  
GetFn (*C function*), 386  
MRISetInnerStepper\_SetAccumulatedErrorRe-  
setFn (*C function*), 387  
MRISetInnerStepper\_SetContent (*C function*), 384  
MRISetInnerStepper\_SetEvolveFn (*C function*), 385  
MRISetInnerStepper\_SetFullRhsFn (*C function*), 386  
MRISetInnerStepper\_SetResetFn (*C function*), 386  
MRISetInnerStepper\_SetRTolFn (*C function*), 387  
MRISetPostInnerFn (*C type*), 184  
MRISetPreInnerFn (*C type*), 184  
MRISetPrintAllStats (*C function*), 361  
MRISetReInit (*C function*), 373  
MRISetReset (*C function*), 374  
MRISetResize (*C function*), 374  
MRISetRootInit (*C function*), 339  
MRISetSetCoupling (*C function*), 346  
MRISetSetDeduceImplicitRhs (*C function*), 351  
MRISetSetDefaults (*C function*), 341  
MRISetSetDeltaGammaMax (*C function*), 352  
MRISetSetDenseOrder (*C function*), 342  
MRISetSetDiagnostics (*C function*), 342  
MRISetSetEpsLin (*C function*), 356  
MRISetSetFixedStep (*C function*), 343  
MRISetSetInterpolantDegree (*C function*), 342  
MRISetSetInterpolantType (*C function*), 341  
MRISetSetInterpolateStopTime (*C function*), 344  
MRISetSetJacEvalFrequency (*C function*), 352  
MRISetSetJacFn (*C function*), 353  
MRISetSetJacTimes (*C function*), 354  
MRISetSetJacTimesRhsFn (*C function*), 355  
MRISetSetLinear (*C function*), 347  
MRISetSetLinearSolutionScaling (*C function*), 354  
MRISetSetLinearSolver (*C function*), 338  
MRISetSetLinSysFn (*C function*), 353  
MRISetSetLSetupFrequency (*C function*), 352  
MRISetSetLSNormFactor (*C function*), 356  
MRISetSetMaxHnilWarns (*C function*), 343  
MRISetSetMaxNonlinIters (*C function*), 349  
MRISetSetMaxNumSteps (*C function*), 344

MRISetSetNlsRhsFn (*C function*), 351  
MRISetSetNoInactiveRootWarn (*C function*), 357  
MRISetSetNonlinConvCoef (*C function*), 349  
MRISetSetNonlinCRDown (*C function*), 350  
MRISetSetNonlinear (*C function*), 348  
MRISetSetNonlinearSolver (*C function*), 338  
MRISetSetNonlinRDiv (*C function*), 350  
MRISetSetOrder (*C function*), 346  
MRISetSetPostInnerFn (*C function*), 346  
MRISetSetPreconditioner (*C function*), 355  
MRISetSetPredictorMethod (*C function*), 348  
MRISetSetPreInnerFn (*C function*), 345  
MRISetSetRootDirection (*C function*), 357  
MRISetSetStagePredictFn (*C function*), 350  
MRISetSetStopTime (*C function*), 344  
MRISetSetUserData (*C function*), 345  
MRISetSStolerances (*C function*), 337  
MRISetSVtolerances (*C function*), 337  
MRISetWftolerances (*C function*), 337  
MRISetWriteCoupling (*C function*), 372  
MRISetWriteParameters (*C function*), 372

## N

N\_VAbs (*C function*), 444  
N\_VAddConst (*C function*), 444  
N\_VBufPack (*C function*), 453  
N\_VBufSize (*C function*), 453  
N\_VBufUnpack (*C function*), 453  
N\_VClone (*C function*), 441  
N\_VCloneEmpty (*C function*), 441  
N\_VCloneVectorArray (*C function*), 437  
N\_VCloneVectorArrayEmpty (*C function*), 438  
N\_VCompare (*C function*), 446  
N\_VConst (*C function*), 443  
N\_VConstrMask (*C function*), 446  
N\_VConstrMaskLocal (*C function*), 451  
N\_VConstVectorArray (*C function*), 448  
N\_VCopyFromDevice\_Cuda (*C function*), 475  
N\_VCopyFromDevice\_Hip (*C function*), 480  
N\_VCopyFromDevice\_OpenMPDEV (*C function*), 495  
N\_VCopyFromDevice\_Raja (*C function*), 489  
N\_VCopyFromDevice\_Sycl (*C++ function*), 484  
N\_VCopyOps (*C function*), 440  
N\_VCopyToDevice\_Cuda (*C function*), 474  
N\_VCopyToDevice\_Hip (*C function*), 479  
N\_VCopyToDevice\_OpenMPDEV (*C function*), 495  
N\_VCopyToDevice\_Raja (*C function*), 489  
N\_VCopyToDevice\_Sycl (*C++ function*), 484  
N\_VDestroy (*C function*), 442  
N\_VDestroyVectorArray (*C function*), 438  
N\_VDiv (*C function*), 444  
N\_VDotProd (*C function*), 444  
N\_VDotProdLocal (*C function*), 450  
N\_VDotProdMulti (*C function*), 447

- N\_VDotProdMultiAllReduce (C function), 452  
 N\_VDotProdMultiLocal (C function), 452  
 N\_Vector (C type), 433  
 N\_Vector\_ID (C enum), 440  
 N\_Vector\_ID (class in sundials4py.core), 848  
 N\_Vector\_Ops (C type), 433  
 N\_VEnableConstVectorArray\_Cuda (C function), 475  
 N\_VEnableConstVectorArray\_Hip (C function), 480  
 N\_VEnableConstVectorArray\_ManyVector (C function), 500  
 N\_VEnableConstVectorArray\_MPIManyVector (C function), 504  
 N\_VEnableConstVectorArray\_OpenMP (C function), 464  
 N\_VEnableConstVectorArray\_OpenMPDEV (C function), 496  
 N\_VEnableConstVectorArray\_Parallel (C function), 460  
 N\_VEnableConstVectorArray\_ParHyp (C function), 470  
 N\_VEnableConstVectorArray\_Petsc (C function), 472  
 N\_VEnableConstVectorArray\_Pthreads (C function), 467  
 N\_VEnableConstVectorArray\_Raja (C function), 490  
 N\_VEnableConstVectorArray\_Serial (C function), 457  
 N\_VEnableConstVectorArray\_Sycl (C++ function), 486  
 N\_VEnableDotProdMulti\_Cuda (C function), 475  
 N\_VEnableDotProdMulti\_Hip (C function), 480  
 N\_VEnableDotProdMulti\_ManyVector (C function), 500  
 N\_VEnableDotProdMulti\_MPIManyVector (C function), 504  
 N\_VEnableDotProdMulti\_OpenMP (C function), 464  
 N\_VEnableDotProdMulti\_OpenMPDEV (C function), 496  
 N\_VEnableDotProdMulti\_Parallel (C function), 460  
 N\_VEnableDotProdMulti\_ParHyp (C function), 470  
 N\_VEnableDotProdMulti\_Petsc (C function), 472  
 N\_VEnableDotProdMulti\_Pthreads (C function), 467  
 N\_VEnableDotProdMulti\_Serial (C function), 457  
 N\_VEnableFusedOps\_Cuda (C function), 475  
 N\_VEnableFusedOps\_Hip (C function), 480  
 N\_VEnableFusedOps\_ManyVector (C function), 500  
 N\_VEnableFusedOps\_MPIManyVector (C function), 503  
 N\_VEnableFusedOps\_OpenMP (C function), 463  
 N\_VEnableFusedOps\_OpenMPDEV (C function), 496  
 N\_VEnableFusedOps\_Parallel (C function), 460  
 N\_VEnableFusedOps\_ParHyp (C function), 469  
 N\_VEnableFusedOps\_Petsc (C function), 471  
 N\_VEnableFusedOps\_Pthreads (C function), 467  
 N\_VEnableFusedOps\_Raja (C function), 490  
 N\_VEnableFusedOps\_Serial (C function), 456  
 N\_VEnableFusedOps\_Sycl (C++ function), 485  
 N\_VEnableLinearCombination\_Cuda (C function), 475  
 N\_VEnableLinearCombination\_Hip (C function), 480  
 N\_VEnableLinearCombination\_ManyVector (C function), 500  
 N\_VEnableLinearCombination\_MPIManyVector (C function), 504  
 N\_VEnableLinearCombination\_OpenMP (C function), 464  
 N\_VEnableLinearCombination\_OpenMPDEV (C function), 496  
 N\_VEnableLinearCombination\_Parallel (C function), 460  
 N\_VEnableLinearCombination\_ParHyp (C function), 469  
 N\_VEnableLinearCombination\_Petsc (C function), 471  
 N\_VEnableLinearCombination\_Pthreads (C function), 467  
 N\_VEnableLinearCombination\_Raja (C function), 490  
 N\_VEnableLinearCombination\_Serial (C function), 457  
 N\_VEnableLinearCombination\_Sycl (C++ function), 485  
 N\_VEnableLinearCombinationVectorArray\_Cuda (C function), 476  
 N\_VEnableLinearCombinationVectorArray\_Hip (C function), 481  
 N\_VEnableLinearCombinationVectorArray\_OpenMP (C function), 464  
 N\_VEnableLinearCombinationVectorArray\_OpenMPDEV (C function), 496  
 N\_VEnableLinearCombinationVectorArray\_Parallel (C function), 461  
 N\_VEnableLinearCombinationVectorArray\_ParHyp (C function), 470  
 N\_VEnableLinearCombinationVectorArray\_Petsc (C function), 472  
 N\_VEnableLinearCombinationVectorArray\_Pthreads (C function), 468  
 N\_VEnableLinearCombinationVectorArray\_Raja (C function), 490  
 N\_VEnableLinearCombinationVectorArray\_Serial (C function), 457  
 N\_VEnableLinearCombinationVectorArray\_Sycl (C++ function), 486  
 N\_VEnableLinearSumVectorArray\_Cuda (C function), 475  
 N\_VEnableLinearSumVectorArray\_Hip (C function), 480

<code>N_VEnableLinearSumVectorArray_ManyVector</code> (C function), 500	<code>N_VEnableScaleAddMultiVectorArray_Pthreads</code> (C function), 468
<code>N_VEnableLinearSumVectorArray_MPIManyVector</code> (C function), 504	<code>N_VEnableScaleAddMultiVectorArray_Raja</code> (C function), 490
<code>N_VEnableLinearSumVectorArray_OpenMP</code> (C function), 464	<code>N_VEnableScaleAddMultiVectorArray_Serial</code> (C function), 457
<code>N_VEnableLinearSumVectorArray_OpenMPDEV</code> (C function), 496	<code>N_VEnableScaleAddMultiVectorArray_Sycl</code> (C++ function), 486
<code>N_VEnableLinearSumVectorArray_Parallel</code> (C function), 460	<code>N_VEnableScaleVectorArray_Cuda</code> (C function), 475
<code>N_VEnableLinearSumVectorArray_ParHyp</code> (C function), 470	<code>N_VEnableScaleVectorArray_Hip</code> (C function), 480
<code>N_VEnableLinearSumVectorArray_Petsc</code> (C function), 472	<code>N_VEnableScaleVectorArray_ManyVector</code> (C function), 500
<code>N_VEnableLinearSumVectorArray_Pthreads</code> (C function), 467	<code>N_VEnableScaleVectorArray_MPIManyVector</code> (C function), 504
<code>N_VEnableLinearSumVectorArray_Raja</code> (C function), 490	<code>N_VEnableScaleVectorArray_OpenMP</code> (C function), 464
<code>N_VEnableLinearSumVectorArray_Serial</code> (C function), 457	<code>N_VEnableScaleVectorArray_OpenMPDEV</code> (C function), 496
<code>N_VEnableLinearSumVectorArray_Sycl</code> (C++ function), 485	<code>N_VEnableScaleVectorArray_Parallel</code> (C function), 460
<code>N_VEnableScaleAddMulti_Cuda</code> (C function), 475	<code>N_VEnableScaleVectorArray_ParHyp</code> (C function), 470
<code>N_VEnableScaleAddMulti_Hip</code> (C function), 480	<code>N_VEnableScaleVectorArray_Petsc</code> (C function), 472
<code>N_VEnableScaleAddMulti_ManyVector</code> (C function), 500	<code>N_VEnableScaleVectorArray_Pthreads</code> (C function), 467
<code>N_VEnableScaleAddMulti_MPIManyVector</code> (C function), 504	<code>N_VEnableScaleVectorArray_Raja</code> (C function), 490
<code>N_VEnableScaleAddMulti_OpenMP</code> (C function), 464	<code>N_VEnableScaleVectorArray_Serial</code> (C function), 457
<code>N_VEnableScaleAddMulti_OpenMPDEV</code> (C function), 496	<code>N_VEnableScaleVectorArray_Sycl</code> (C++ function), 485
<code>N_VEnableScaleAddMulti_Parallel</code> (C function), 460	<code>N_VEnableWrmsNormMaskVectorArray_Cuda</code> (C function), 475
<code>N_VEnableScaleAddMulti_ParHyp</code> (C function), 469	<code>N_VEnableWrmsNormMaskVectorArray_Hip</code> (C function), 480
<code>N_VEnableScaleAddMulti_Petsc</code> (C function), 472	<code>N_VEnableWrmsNormMaskVectorArray_ManyVector</code> (C function), 500
<code>N_VEnableScaleAddMulti_Pthreads</code> (C function), 467	<code>N_VEnableWrmsNormMaskVectorArray_MPI-ManyVector</code> (C function), 504
<code>N_VEnableScaleAddMulti_Raja</code> (C function), 490	<code>N_VEnableWrmsNormMaskVectorArray_OpenMP</code> (C function), 464
<code>N_VEnableScaleAddMulti_Serial</code> (C function), 457	<code>N_VEnableWrmsNormMaskVectorArray_OpenMPDEV</code> (C function), 496
<code>N_VEnableScaleAddMulti_Sycl</code> (C++ function), 485	<code>N_VEnableWrmsNormMaskVectorArray_Parallel</code> (C function), 460
<code>N_VEnableScaleAddMultiVectorArray_Cuda</code> (C function), 475	<code>N_VEnableWrmsNormMaskVectorArray_ParHyp</code> (C function), 470
<code>N_VEnableScaleAddMultiVectorArray_Hip</code> (C function), 480	<code>N_VEnableWrmsNormMaskVectorArray_Petsc</code> (C function), 472
<code>N_VEnableScaleAddMultiVectorArray_OpenMP</code> (C function), 464	<code>N_VEnableWrmsNormMaskVectorArray_Pthreads</code> (C function), 467
<code>N_VEnableScaleAddMultiVectorArray_OpenMPDEV</code> (C function), 496	<code>N_VEnableWrmsNormMaskVectorArray_Serial</code> (C function), 457
<code>N_VEnableScaleAddMultiVectorArray_Parallel</code> (C function), 461	<code>N_VEnableWrmsNormVectorArray_Cuda</code> (C function),
<code>N_VEnableScaleAddMultiVectorArray_ParHyp</code> (C function), 470	
<code>N_VEnableScaleAddMultiVectorArray_Petsc</code> (C function), 472	



- 475  
N\_VEnableWrmsNormVectorArray\_Hip (C function), 480  
N\_VEnableWrmsNormVectorArray\_ManyVector (C function), 500  
N\_VEnableWrmsNormVectorArray\_MPIManyVector (C function), 504  
N\_VEnableWrmsNormVectorArray\_OpenMP (C function), 464  
N\_VEnableWrmsNormVectorArray\_OpenMPDEV (C function), 496  
N\_VEnableWrmsNormVectorArray\_Parallel (C function), 460  
N\_VEnableWrmsNormVectorArray\_ParHyp (C function), 470  
N\_VEnableWrmsNormVectorArray\_Petsc (C function), 472  
N\_VEnableWrmsNormVectorArray\_Pthreads (C function), 467  
N\_VEnableWrmsNormVectorArray\_Serial (C function), 457  
N\_VFreeEmpty (C function), 439  
N\_VGetArrayPointer (C function), 442  
N\_VGetArrayPointer\_MPIPlusX (C function), 505  
N\_VGetCommunicator (C function), 443  
N\_VGetDeviceArrayPointer (C function), 442  
N\_VGetDeviceArrayPointer\_Cuda (C function), 473  
N\_VGetDeviceArrayPointer\_Hip (C function), 478  
N\_VGetDeviceArrayPointer\_OpenMPDEV (C function), 495  
N\_VGetDeviceArrayPointer\_Raja (C function), 489  
N\_VGetDeviceArrayPointer\_Sycl (C++ function), 484  
N\_VGetHostArrayPointer\_Cuda (C function), 473  
N\_VGetHostArrayPointer\_Hip (C function), 478  
N\_VGetHostArrayPointer\_OpenMPDEV (C function), 495  
N\_VGetHostArrayPointer\_Raja (C function), 489  
N\_VGetHostArrayPointer\_Sycl (C++ function), 484  
N\_VGetLength (C function), 443  
N\_VGetLocalLength (C function), 443  
N\_VGetLocalLength\_MPIPlusX (C function), 505  
N\_VGetLocalLength\_Parallel (C function), 460  
N\_VGetLocalVector\_MPIPlusX (C function), 505  
N\_VGetNumSubvectors\_ManyVector (C function), 500  
N\_VGetNumSubvectors\_MPIManyVector (C function), 503  
N\_VGetSubvector\_ManyVector (C function), 499  
N\_VGetSubvector\_MPIManyVector (C function), 503  
N\_VGetSubvectorArrayPointer\_ManyVector (C function), 499  
N\_VGetSubvectorArrayPointer\_MPIManyVector (C function), 503  
N\_VGetSubvectorLocalLength\_ManyVector (C function), 499  
N\_VGetSubvectorLocalLength\_MPIManyVector (C function), 503  
N\_VGetVecAtIndexVectorArray (C function), 438  
N\_VGetVector\_ParHyp (C function), 469  
N\_VGetVector\_Petsc (C function), 471  
N\_VGetVector\_Trilinos (C++ function), 497  
N\_VGetVectorID (C function), 441  
N\_VInv (C function), 444  
N\_VInvTest (C function), 446  
N\_VInvTestLocal (C function), 451  
N\_VIsManagedMemory\_Cuda (C function), 473  
N\_VIsManagedMemory\_Hip (C function), 479  
N\_VIsManagedMemory\_Raja (C function), 489  
N\_VIsManagedMemory\_Sycl (C++ function), 484  
N\_VL1Norm (C function), 446  
N\_VL1NormLocal (C function), 450  
N\_VLinearCombination (C function), 447  
N\_VLinearCombinationVectorArray (C function), 449  
N\_VLinearSum (C function), 443  
N\_VLinearSumVectorArray (C function), 448  
N\_VMake\_Cuda (C function), 474  
N\_VMake\_Hip (C function), 479  
N\_VMake\_MPIManyVector (C function), 503  
N\_VMake\_MPIPlusX (C function), 505  
N\_VMake\_OpenMP (C function), 463  
N\_VMake\_OpenMPDEV (C function), 495  
N\_VMake\_Parallel (C function), 460  
N\_VMake\_ParHyp (C function), 469  
N\_VMake\_Petsc (C function), 471  
N\_VMake\_Pthreads (C function), 467  
N\_VMake\_Raja (C function), 489  
N\_VMake\_Serial (C function), 456  
N\_VMake\_Sycl (C++ function), 484  
N\_VMake\_Trilinos (C++ function), 498  
N\_VMakeManaged\_Cuda (C function), 474  
N\_VMakeManaged\_Hip (C function), 479  
N\_VMakeManaged\_Raja (C function), 489  
N\_VMakeManaged\_Sycl (C++ function), 484  
N\_VMakeWithManagedAllocator\_Cuda (C function), 474  
N\_VMaxNorm (C function), 445  
N\_VMaxNormLocal (C function), 450  
N\_VMin (C function), 445  
N\_VMinLocal (C function), 450  
N\_VMinQuotient (C function), 447  
N\_VMinQuotientLocal (C function), 452  
N\_VNew\_Cuda (C function), 474  
N\_VNew\_Hip (C function), 479  
N\_VNew\_ManyVector (C function), 499  
N\_VNew\_MPIManyVector (C function), 502  
N\_VNew\_OpenMP (C function), 463

`N_VNew_OpenMPDEV` (C function), 495  
`N_VNew_Parallel` (C function), 459  
`N_VNew_Pthreads` (C function), 466  
`N_VNew_Raja` (C function), 489  
`N_VNew_Serial` (C function), 456  
`N_VNew_Sycl` (C++ function), 484  
`N_VNewEmpty` (C function), 439  
`N_VNewEmpty_Cuda` (C function), 474  
`N_VNewEmpty_Hip` (C function), 479  
`N_VNewEmpty_OpenMP` (C function), 463  
`N_VNewEmpty_OpenMPDEV` (C function), 495  
`N_VNewEmpty_Parallel` (C function), 460  
`N_VNewEmpty_ParHyp` (C function), 469  
`N_VNewEmpty_Petsc` (C function), 471  
`N_VNewEmpty_Pthreads` (C function), 467  
`N_VNewEmpty_Raja` (C function), 489  
`N_VNewEmpty_Serial` (C function), 456  
`N_VNewEmpty_Sycl` (C++ function), 484  
`N_VNewManaged_Cuda` (C function), 474  
`N_VNewManaged_Hip` (C function), 479  
`N_VNewManaged_Raja` (C function), 489  
`N_VNewManaged_Sycl` (C++ function), 484  
`N_VNewVectorArray` (C function), 438  
`N_VNewWithMemHelp_Cuda` (C function), 474  
`N_VNewWithMemHelp_Hip` (C function), 479  
`N_VNewWithMemHelp_Raja` (C function), 489  
`N_VNewWithMemHelp_Sycl` (C++ function), 484  
`N_VPrint` (C function), 453  
`N_VPrint_Cuda` (C function), 475  
`N_VPrint_Hip` (C function), 480  
`N_VPrint_OpenMP` (C function), 463  
`N_VPrint_OpenMPDEV` (C function), 495  
`N_VPrint_Parallel` (C function), 460  
`N_VPrint_ParHyp` (C function), 469  
`N_VPrint_Petsc` (C function), 471  
`N_VPrint_Pthreads` (C function), 467  
`N_VPrint_Raja` (C function), 489  
`N_VPrint_Serial` (C function), 456  
`N_VPrint_Sycl` (C++ function), 485  
`N_VPrintFile` (C function), 453  
`N_VPrintFile_Cuda` (C function), 475  
`N_VPrintFile_Hip` (C function), 480  
`N_VPrintFile_OpenMP` (C function), 463  
`N_VPrintFile_OpenMPDEV` (C function), 495  
`N_VPrintFile_Parallel` (C function), 460  
`N_VPrintFile_ParHyp` (C function), 469  
`N_VPrintFile_Petsc` (C function), 471  
`N_VPrintFile_Pthreads` (C function), 467  
`N_VPrintFile_Raja` (C function), 490  
`N_VPrintFile_Serial` (C function), 456  
`N_VPrintFile_Sycl` (C++ function), 485  
`N_VProd` (C function), 443  
`N_VScale` (C function), 444  
`N_VScaleAddMulti` (C function), 447  
`N_VScaleAddMultiVectorArray` (C function), 449  
`N_VScaleVectorArray` (C function), 448  
`N_VSetArrayPointer` (C function), 442  
`N_VSetArrayPointer_MPIPlusX` (C function), 506  
`N_VSetDeviceArrayPointer_Sycl` (C++ function), 484  
`N_VSetHostArrayPointer_Sycl` (C++ function), 484  
`N_VSetKernelExecPolicy_Cuda` (C function), 474  
`N_VSetKernelExecPolicy_Hip` (C function), 479  
`N_VSetKernelExecPolicy_Sycl` (C++ function), 485  
`N_VSetSubvectorArrayPointer_ManyVector` (C function), 500  
`N_VSetSubvectorArrayPointer_MPIManyVector` (C function), 503  
`N_VSetVecAtIndexVectorArray` (C function), 439  
`N_VSpace` (C function), 442  
`N_VWL2Norm` (C function), 445  
`N_VWrmsNorm` (C function), 445  
`N_VWrmsNormMask` (C function), 445  
`N_VWrmsNormMaskVectorArray` (C function), 449  
`N_VWrmsNormVectorArray` (C function), 448  
`N_VWSqrSumLocal` (C function), 451  
`N_VWSqrSumMaskLocal` (C function), 451  
Newton linear system, 30  
Newton update, 30  
Newton's method, 30  
`NV_COMM_P` (C macro), 459  
`NV_CONTENT_OMP` (C macro), 462  
`NV_CONTENT_OMPDEV` (C macro), 494  
`NV_CONTENT_P` (C macro), 458  
`NV_CONTENT_PT` (C macro), 465  
`NV_CONTENT_S` (C macro), 455  
`NV_DATA_DEV_OMPDEV` (C macro), 494  
`NV_DATA_HOST_OMPDEV` (C macro), 494  
`NV_DATA_OMP` (C macro), 462  
`NV_DATA_P` (C macro), 458  
`NV_DATA_PT` (C macro), 466  
`NV_DATA_S` (C macro), 455  
`NV_GLOBLLENGTH_P` (C macro), 459  
`NV_Ith_OMP` (C macro), 463  
`NV_Ith_P` (C macro), 459  
`NV_Ith_PT` (C macro), 466  
`NV_Ith_S` (C macro), 456  
`NV_LENGTH_OMP` (C macro), 462  
`NV_LENGTH_OMPDEV` (C macro), 495  
`NV_LENGTH_PT` (C macro), 466  
`NV_LENGTH_S` (C macro), 455  
`NV_LOCLENGTH_P` (C macro), 459  
`NV_NUM_THREADS_OMP` (C macro), 462  
`NV_NUM_THREADS_PT` (C macro), 466  
`NV_OWN_DATA_OMP` (C macro), 462  
`NV_OWN_DATA_OMPDEV` (C macro), 494  
`NV_OWN_DATA_P` (C macro), 458  
`NV_OWN_DATA_PT` (C macro), 465

NV\_OWN\_DATA\_S (C macro), 455

## O

optional input

generic linear solver interface  
(ARKODE), 117

Jacobian update frequency (ARKODE), 119

linear solver setup frequency (ARKODE),  
118

preconditioner update frequency  
(ARKODE), 119

## P

print\_log() (in module logs), 59

## R

residual weight vector, 23

## S

SM\_COLS\_B (C macro), 532

SM\_COLS\_D (C macro), 517

SM\_COLUMN\_B (C macro), 532

SM\_COLUMN\_D (C macro), 517

SM\_COLUMN\_ELEMENT\_B (C macro), 532

SM\_COLUMNS\_B (C macro), 531

SM\_COLUMNS\_D (C macro), 517

SM\_COLUMNS\_S (C macro), 541

SM\_CONTENT\_B (C macro), 529

SM\_CONTENT\_D (C macro), 516

SM\_CONTENT\_S (C macro), 539

SM\_DATA\_B (C macro), 532

SM\_DATA\_D (C macro), 517

SM\_DATA\_S (C macro), 541

SM\_ELEMENT\_B (C macro), 532

SM\_ELEMENT\_D (C macro), 518

SM\_INDEXPTRS\_S (C macro), 541

SM\_INDEXVALS\_S (C macro), 541

SM\_LBAND\_B (C macro), 531

SM\_LDATA\_B (C macro), 531

SM\_LDATA\_D (C macro), 517

SM\_LDIM\_B (C macro), 531

SM\_NNZ\_S (C macro), 541

SM\_NP\_S (C macro), 541

SM\_ROWS\_B (C macro), 529

SM\_ROWS\_D (C macro), 517

SM\_ROWS\_S (C macro), 539

SM\_SPARSETYPE\_S (C macro), 541

SM\_SUBAND\_B (C macro), 531

SM\_UBAND\_B (C macro), 531

SplittingStep user main program, 392

SplittingStepCoefficients (C type), 396

SplittingStepCoefficients\_Alloc (C function),  
401

SplittingStepCoefficients\_Copy (C function), 401  
SplittingStepCoefficients\_Create (C function),  
401

SplittingStepCoefficients\_Destroy (C function),  
402

SplittingStepCoefficients\_IDToName (C func-  
tion), 399

SplittingStepCoefficients\_LieTrotter (C func-  
tion), 399

SplittingStepCoefficients\_LoadCoefficients  
(C function), 398

SplittingStepCoefficients\_LoadCoefficients-  
ByName (C function), 398

SplittingStepCoefficients\_Parallel (C func-  
tion), 399

SplittingStepCoefficients\_Strang (C function),  
399

SplittingStepCoefficients\_SuzukiFractal (C  
function), 400

SplittingStepCoefficients\_SymmetricParallel  
(C function), 399

SplittingStepCoefficients\_ThirdOrderSuzuki  
(C function), 400

SplittingStepCoefficients\_TripleJump (C func-  
tion), 400

SplittingStepCoefficients\_Write (C function),  
402

SplittingStepCoefficientsMem (C struct), 397

SplittingStepCoefficientsMem (class in *sundi-  
als4py.arkode*), 883

SplittingStepCoefficientsMem.alpha (C mem-  
ber), 397

SplittingStepCoefficientsMem.beta (C member),  
397

SplittingStepCoefficientsMem.order (C mem-  
ber), 397

SplittingStepCoefficientsMem.partitions (C  
member), 397

SplittingStepCoefficientsMem.sequential\_-  
methods (C member), 397

SplittingStepCoefficientsMem.stages (C mem-  
ber), 397

SplittingStepCreate (C function), 393

SplittingStepGetNumEvolves (C function), 395

SplittingStepReInit (C function), 396

SplittingStepSetCoefficients (C function), 394

SPRKStepClearStopTime (C function), 408

SPRKStepCreate (C function), 403

SPRKStepEvolve (C function), 405

SPRKStepFree (C function), 404

SPRKStepGetCurrentMethod (C function), 415

SPRKStepGetCurrentState (C function), 413

SPRKStepGetCurrentStep (C function), 413

SPRKStepGetCurrentTime (C function), 413

SPRKStepGetDky (C function), 411	
SPRKStepGetLastStep (C function), 412	
SPRKStepGetNumGEvals (C function), 416	
SPRKStepGetNumRhsEvals (C function), 415	
SPRKStepGetNumStepAttempts (C function), 415	
SPRKStepGetNumSteps (C function), 412	
SPRKStepGetReturnFlagName (C function), 414	
SPRKStepGetRootInfo (C function), 416	
SPRKStepGetStepStats (C function), 413	
SPRKStepGetUserData (C function), 415	
SPRKStepPrintAllStats (C function), 414	
SPRKStepReInit (C function), 417	
SPRKStepReset (C function), 418	
SPRKStepRootInit (C function), 404	
SPRKStepSetDefaults (C function), 406	
SPRKStepSetFixedStep (C function), 407	
SPRKStepSetInterpolantDegree (C function), 406	
SPRKStepSetInterpolantType (C function), 406	
SPRKStepSetMaxNumSteps (C function), 407	
SPRKStepSetMethod (C function), 409	
SPRKStepSetMethodName (C function), 410	
SPRKStepSetNoInactiveRootWarn (C function), 411	
SPRKStepSetOrder (C function), 409	
SPRKStepSetRootDirection (C function), 411	
SPRKStepSetStopTime (C function), 408	
SPRKStepSetUseCompensatedSums (C function), 410	
SPRKStepSetUserData (C function), 408	
SPRKStepWriteParameters (C function), 416	
SUN_ADAPTCONTROLLER_H (C enumerator), 665	
SUN_ADAPTCONTROLLER_H	(sundials4py.core.SUNAdaptController_Type attribute), 848
SUN_ADAPTCONTROLLER_MRI_H_TOL (C enumerator), 665	
SUN_ADAPTCONTROLLER_MRI_H_TOL	(sundials4py.core.SUNAdaptController_Type attribute), 848
SUN_ADAPTCONTROLLER_NONE (C enumerator), 665	
SUN_ADAPTCONTROLLER_NONE	(sundials4py.core.SUNAdaptController_Type attribute), 848
SUN_CLASSICAL_GS	(sundials4py.core.SUNGramSchmidtType attribute), 850
SUN_COMM_NULL (C macro), 48	
SUN_ERR_ADJOINT_STEPPERFAILED	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_ADJOINT_STEPPERINVALIDSTOP	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_ARG_CORRUPT (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_ARG_DIMSMISMATCH	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_ARG_INCOMPATIBLE	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_ARG_OUTOFRANGE	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_ARG_WRONGTYPE	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_CHECKPOINT_MISMATCH	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_CHECKPOINT_NOT_FOUND	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_CORRUPT (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_DATANODE_NODENOTFOUND	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_DESTROY_FAIL	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_EXT_FAIL (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_FILE_OPEN (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_GENERIC (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_MALLOC_FAIL (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_MAXIMUM (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_MEM_FAIL (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_MINIMUM (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_MPI_FAIL (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_NOT_IMPLEMENTED	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_OP_FAIL (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_OUTOFRANGE (sundials4py.core.SUNErrCode attribute), 849	
SUN_ERR_PROFILER_MAPFULL	(sundials4py.core.SUNErrCode attribute), 849
SUN_ERR_PROFILER_MAPGET	(sundials4py.core.SUNErrCode attribute), 850
SUN_ERR_PROFILER_MAPINSERT	(sundials4py.core.SUNErrCode attribute), 850
SUN_ERR_PROFILER_MAPKEYNOTFOUND	(sundials4py.core.SUNErrCode attribute), 850
SUN_ERR_PROFILER_MAPSORT	(sundials4py.core.SUNErrCode attribute), 850
SUN_ERR_SUNCTX_CORRUPT	(sundials4py.core.SUNErrCode attribute), 850
SUN_ERR_UNKNOWN (sundials4py.core.SUNErrCode attribute), 850	
SUN_ERR_UNREACHABLE (sundials4py.core.SUNErrCode attribute), 850	



SUN_ERR_USER_FCN_FAIL	( <i>sundials4py.core.SUNErrCode</i> attribute), 850	SUNAdaptController_H0211 (C function), 676
SUN_FULLLRHS_END	( <i>sundials4py.core.SUNFullRhsMode</i> attribute), 850	SUNAdaptController_H0321 (C function), 676
SUN_FULLLRHS_OTHER	( <i>sundials4py.core.SUNFullRhsMode</i> attribute), 850	SUNAdaptController_H211 (C function), 676
SUN_FULLLRHS_START	( <i>sundials4py.core.SUNFullRhsMode</i> attribute), 850	SUNAdaptController_H312 (C function), 676
SUN_LOGLEVEL_ALL	( <i>sundials4py.core.SUNLogLevel</i> attribute), 851	SUNAdaptController_I (C function), 673
SUN_LOGLEVEL_DEBUG	( <i>sundials4py.core.SUNLogLevel</i> attribute), 851	SUNAdaptController_ImExGus (C function), 678
SUN_LOGLEVEL_ERROR	( <i>sundials4py.core.SUNLogLevel</i> attribute), 851	SUNAdaptController_ImpGus (C function), 675
SUN_LOGLEVEL_INFO	( <i>sundials4py.core.SUNLogLevel</i> attribute), 851	SUNAdaptController_MRIHTol (C function), 680
SUN_LOGLEVEL_NONE	( <i>sundials4py.core.SUNLogLevel</i> attribute), 851	SUNAdaptController_NewEmpty (C function), 665
SUN_LOGLEVEL_WARNING	( <i>sundials4py.core.SUNLogLevel</i> attribute), 851	SUNAdaptController_Ops (C type), 664
SUN_MODIFIED_GS	( <i>sundials4py.core.SUNGramSchmidtType</i> attribute), 850	SUNAdaptController_PI (C function), 672
SUN_OUTPUTFORMAT_CSV	(C enumerator), 47	SUNAdaptController_PID (C function), 672
SUN_OUTPUTFORMAT_CSV	( <i>sundials4py.core.SUNOutputFormat</i> attribute), 852	SUNAdaptController_Reset (C function), 667
SUN_OUTPUTFORMAT_TABLE	(C enumerator), 47	SUNAdaptController_SetDefaults (C function), 668
SUN_OUTPUTFORMAT_TABLE	( <i>sundials4py.core.SUNOutputFormat</i> attribute), 852	SUNAdaptController_SetErrorBias (C function), 668
SUN_PREC_BOTH	( <i>sundials4py.core.SUNPrecType</i> attribute), 852	SUNAdaptController_SetOptions (C function), 667
SUN_PREC_LEFT	( <i>sundials4py.core.SUNPrecType</i> attribute), 852	SUNAdaptController_SetParams_ExpGus (C function), 674
SUN_PREC_NONE	( <i>sundials4py.core.SUNPrecType</i> attribute), 852	SUNAdaptController_SetParams_I (C function), 673
SUN_PREC_RIGHT	( <i>sundials4py.core.SUNPrecType</i> attribute), 852	SUNAdaptController_SetParams_ImExGus (C function), 678
SUN_SUCCESS	( <i>sundials4py.core.SUNErrCode</i> attribute), 850	SUNAdaptController_SetParams_ImpGus (C function), 675
SUNAbortErrHandlerFn (C function), 55		SUNAdaptController_SetParams_MRIHTol (C function), 680
SUNAdaptController (C type), 663		SUNAdaptController_SetParams_PI (C function), 673
SUNAdaptController_Destroy (C function), 666		SUNAdaptController_SetParams_PID (C function), 672
SUNAdaptController_DestroyEmpty (C function), 665		SUNAdaptController_SetParams_Soderlind (C function), 671
SUNAdaptController_EstimateStep (C function), 666		SUNAdaptController_Soderlind (C function), 671
SUNAdaptController_EstimateStepTol (C function), 666		SUNAdaptController_Space (C function), 669
SUNAdaptController_ExpGus (C function), 674		SUNAdaptController_Type (C enum), 665
SUNAdaptController_GetType (C function), 665		SUNAdaptController_Type (class in <i>sundials4py.core</i> ), 848
		SUNAdaptController_UpdateH (C function), 669
		SUNAdaptController_UpdateMRIHTol (C function), 669
		SUNAdaptController_Write (C function), 668
		SUNAdaptControllerContent_MRIHTol_ (C struct), 679
		SUNAdaptControllerContent_MRIHTol_ (class in <i>sundials4py.core</i> ), 848
		SUNAdaptControllerContent_MRIHTol_.HControl (C member), 680
		SUNAdaptControllerContent_MRIHTol_.inner_.max_relch (C member), 680
		SUNAdaptControllerContent_MRIHTol_.inner_.max_tolfac (C member), 680
		SUNAdaptControllerContent_MRIHTol_.inner_.min_tolfac (C member), 680

- SUNAdaptControllerContent\_MRIHTol\_.TolControl (C member), 680
- SUNAdjointCheckpointScheme (C type), 698
- SUNAdjointCheckpointScheme\_ (class in sundials4py.core), 848
- SUNAdjointCheckpointScheme\_Create\_Fixed (C function), 703
- SUNAdjointCheckpointScheme\_Destroy (C function), 700
- SUNAdjointCheckpointScheme\_EnableDense (C function), 700
- SUNAdjointCheckpointScheme\_GetContent (C function), 702
- SUNAdjointCheckpointScheme\_InsertVector (C function), 699
- SUNAdjointCheckpointScheme\_LoadVector (C function), 699
- SUNAdjointCheckpointScheme\_NeedsSaving (C function), 698
- SUNAdjointCheckpointScheme\_NewEmpty (C function), 698
- SUNAdjointCheckpointScheme\_SetContent (C function), 702
- SUNAdjointCheckpointScheme\_SetDestroyFn (C function), 701
- SUNAdjointCheckpointScheme\_SetEnableDenseFn (C function), 701
- SUNAdjointCheckpointScheme\_SetInsertVectorFn (C function), 701
- SUNAdjointCheckpointScheme\_SetLoadVectorFn (C function), 701
- SUNAdjointCheckpointScheme\_SetNeedsSavingFn (C function), 700
- SUNAdjointCheckpointScheme\_DestroyFn (C type), 700
- SUNAdjointCheckpointScheme\_EnableDenseFn (C type), 700
- SUNAdjointCheckpointScheme\_InsertVectorFn (C type), 700
- SUNAdjointCheckpointScheme\_LoadVectorFn (C type), 700
- SUNAdjointCheckpointScheme\_NeedsSavingFn (C type), 700
- SUNAdjointStepper (C type), 694
- SUNAdjointStepper\_ (class in sundials4py.core), 848
- SUNAdjointStepper\_Create (C function), 695
- SUNAdjointStepper\_Evolve (C function), 696
- SUNAdjointStepper\_GetNumRecompute (C function), 697
- SUNAdjointStepper\_GetNumSteps (C function), 697
- SUNAdjointStepper\_OneStep (C function), 696
- SUNAdjointStepper\_PrintAllStats (C function), 697
- SUNAdjointStepper\_RecomputeFwd (C function), 696
- SUNAdjointStepper\_ReInit (C function), 695
- SUNAdjointStepper\_SetUserData (C function), 696
- SUNAdjRhsFn (C type), 697
- SUNATimesFn (C type), 562
- SUNBandMatrix (C function), 533
- SUNBandMatrix\_Cols (C function), 533
- SUNBandMatrix\_Column (C function), 534
- SUNBandMatrix\_Columns (C function), 533
- SUNBandMatrix\_Data (C function), 533
- SUNBandMatrix\_LData (C function), 533
- SUNBandMatrix\_LDim (C function), 533
- SUNBandMatrix\_LowerBandwidth (C function), 533
- SUNBandMatrix\_Print (C function), 533
- SUNBandMatrix\_Rows (C function), 533
- SUNBandMatrix\_StoredUpperBandwidth (C function), 533
- SUNBandMatrix\_UpperBandwidth (C function), 533
- SUNBandMatrixStorage (C function), 533
- sunbooleantype (C type), 47
- SUNBraidApp\_FreeEmpty (C function), 272
- SUNBraidApp\_GetVecImpl (C function), 272
- SUNBraidApp\_NewEmpty (C function), 272
- SUNBraidVector (C type), 273
- SUNBraidVector\_BufPack (C function), 275
- SUNBraidVector\_BufSize (C function), 275
- SUNBraidVector\_BufUnpack (C function), 276
- SUNBraidVector\_Clone (C function), 274
- SUNBraidVector\_Free (C function), 274
- SUNBraidVector\_GetNVector (C function), 273
- SUNBraidVector\_New (C function), 273
- SUNBraidVector\_SpatialNorm (C function), 275
- SUNBraidVector\_Sum (C function), 274
- SUNComm (C type), 48
- SUNContext (C type), 48
- SUNContext\_ (class in sundials4py.core), 848
- SUNContext\_ClearErrHandlers (C function), 50
- SUNContext\_Create (C function), 48
- SUNContext\_Free (C function), 49
- SUNContext\_GetLastError (C function), 49
- SUNContext\_GetLogger (C function), 50
- SUNContext\_GetProfiler (C function), 50
- SUNContext\_PeekLastError (C function), 49
- SUNContext\_PopErrHandler (C function), 50
- SUNContext\_PushErrHandler (C function), 49
- SUNContext\_SetLogger (C function), 50
- SUNContext\_SetProfiler (C function), 50
- suncountertype (C type), 47
- SUNCudaBlockReduceAtomicExecPolicy (C++ function), 477
- SUNCudaBlockReduceExecPolicy (C++ function), 477
- SUNCudaExecPolicy (C++ type), 476
- SUNCudaGridStrideExecPolicy (C++ function), 477
- SUNCudaThreadDirectExecPolicy (C++ function), 477

---

SUNDataIOMode (*C enum*), 698  
 SUNDataIOMode (*class in sundials4py.core*), 848  
 SUNDataIOMode.SUNDATAIOMODE\_INMEM (*C enumerator*), 698  
 SUNDATAIOMODE\_INMEM (*sundials4py.core.SUNDataIOMode attribute*), 849  
 SUNDenseMatrix (*C function*), 518  
 SUNDenseMatrix\_Cols (*C function*), 518  
 SUNDenseMatrix\_Column (*C function*), 518  
 SUNDenseMatrix\_Columns (*C function*), 518  
 SUNDenseMatrix\_Data (*C function*), 518  
 SUNDenseMatrix\_LData (*C function*), 518  
 SUNDenseMatrix\_Print (*C function*), 518  
 SUNDenseMatrix\_Rows (*C function*), 518  
 sundials4py.arkode module, 878  
 sundials4py.core module, 848  
 sundials::cuda::ExecPolicy (*C++ class*), 476  
 sundials::cuda::ExecPolicy::~~ExecPolicy (*C++ function*), 476  
 sundials::cuda::ExecPolicy::atomic (*C++ function*), 476  
 sundials::cuda::ExecPolicy::blockSize (*C++ function*), 476  
 sundials::cuda::ExecPolicy::clone (*C++ function*), 476  
 sundials::cuda::ExecPolicy::clone\_new\_stream (*C++ function*), 476  
 sundials::cuda::ExecPolicy::ExecPolicy (*C++ function*), 476  
 sundials::cuda::ExecPolicy::gridSize (*C++ function*), 476  
 sundials::cuda::ExecPolicy::stream (*C++ function*), 476  
 sundials::ginkgo::BatchLinearSolver (*C++ class*), 615  
 sundials::ginkgo::BatchLinearSolver::~~BatchLinearSolver (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::AvgNumIters (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::BatchLinearSolver (*C++ function*), 615–617  
 sundials::ginkgo::BatchLinearSolver::get (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::GkoExec (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::GkoFactory (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::GkoSolvers (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::operator SUNLinearSolver (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::operator= (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::SetScalingMode (*C++ function*), 618  
 sundials::ginkgo::BatchLinearSolver::SetScalingVectors (*C++ function*), 618  
 sundials::ginkgo::BatchLinearSolver::Setup (*C++ function*), 618  
 sundials::ginkgo::BatchLinearSolver::Solve (*C++ function*), 618  
 sundials::ginkgo::BatchLinearSolver::StddevNumIters (*C++ function*), 617  
 sundials::ginkgo::BatchLinearSolver::SumAvgNumIters (*C++ function*), 618  
 sundials::ginkgo::BatchMatrix (*C++ class*), 548  
 sundials::ginkgo::BatchMatrix::~~BatchMatrix (*C++ function*), 548  
 sundials::ginkgo::BatchMatrix::BatchMatrix (*C++ function*), 548  
 sundials::ginkgo::BatchMatrix::get (*C++ function*), 549  
 sundials::ginkgo::BatchMatrix::GkoExec (*C++ function*), 548  
 sundials::ginkgo::BatchMatrix::GkoMtx (*C++ function*), 548  
 sundials::ginkgo::BatchMatrix::GkoSize (*C++ function*), 548  
 sundials::ginkgo::BatchMatrix::NumBatches (*C++ function*), 548  
 sundials::ginkgo::BatchMatrix::operator SUNMatrix (*C++ function*), 549  
 sundials::ginkgo::BatchMatrix::operator= (*C++ function*), 548  
 sundials::ginkgo::LinearSolver (*C++ class*), 612  
 sundials::ginkgo::LinearSolver::~~LinearSolver (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::get (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::GkoExec (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::GkoFactory (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::GkoSolver (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::LinearSolver (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::NumIters (*C++ function*), 613  
 sundials::ginkgo::LinearSolver::operator SUNLinearSolver (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::operator= (*C++ function*), 612  
 sundials::ginkgo::LinearSolver::ResNorm (*C++ function*), 613

`sundials::ginkgo::LinearSolver::Setup` (C++ function), 613  
`sundials::ginkgo::LinearSolver::Solve` (C++ function), 613  
`sundials::hip::ExecPolicy` (C++ class), 481  
`sundials::hip::ExecPolicy::~ExecPolicy` (C++ function), 481  
`sundials::hip::ExecPolicy::atomic` (C++ function), 481  
`sundials::hip::ExecPolicy::blockSize` (C++ function), 481  
`sundials::hip::ExecPolicy::clone` (C++ function), 481  
`sundials::hip::ExecPolicy::clone_new_stream` (C++ function), 481  
`sundials::hip::ExecPolicy::ExecPolicy` (C++ function), 481  
`sundials::hip::ExecPolicy::gridSize` (C++ function), 481  
`sundials::hip::ExecPolicy::stream` (C++ function), 481  
`sundials::sycl::ExecPolicy` (C++ class), 486  
`sundials::sycl::ExecPolicy::~ExecPolicy` (C++ function), 486  
`sundials::sycl::ExecPolicy::blockSize` (C++ function), 486  
`sundials::sycl::ExecPolicy::clone` (C++ function), 486  
`sundials::sycl::ExecPolicy::gridSize` (C++ function), 486  
`SUNDIALS_NVEC_CUDA` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_CUSTOM` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_HIP` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_KOKKOS` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_MANYVECTOR` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_MPIMANYVECTOR` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_MPIPLUSX` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_OPENMP` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_OPENMPDEV` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_PARALLEL` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_PARHYP` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_PETSC` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_PTHREADS` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_RAJA` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_SERIAL` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_SYCL` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALS_NVEC_TRILINOS` (*sundials4py.core.N\_Vector\_ID* attribute), 848  
`SUNDIALSFileClose` (C function), 834  
`SUNDIALSFileOpen` (C function), 834  
`SUNDIALSGetVersion` (C function), 67  
`SUNDIALSGetVersionNumber` (C function), 67  
`SUNDomEigEstimator` (C type), 655  
`SUNDomEigEstimator_` (C struct), 655  
`SUNDomEigEstimator_` (class in *sundials4py.core*), 849  
`SUNDomEigEstimator_.content` (C member), 655  
`SUNDomEigEstimator_.ops` (C member), 655  
`SUNDomEigEstimator_.sunctx` (C member), 655  
`SUNDomEigEstimator_Arnoldi` (C function), 660  
`SUNDomEigEstimator_Destroy` (C function), 651  
`SUNDomEigEstimator_Estimate` (C function), 650  
`SUNDomEigEstimator_FreeEmpty` (C function), 650  
`SUNDomEigEstimator_GetNumATimesCalls` (C function), 654  
`SUNDomEigEstimator_GetNumIters` (C function), 654  
`SUNDomEigEstimator_GetRes` (C function), 654  
`SUNDomEigEstimator_Initialize` (C function), 650  
`SUNDomEigEstimator_NewEmpty` (C function), 650  
`SUNDomEigEstimator_Ops` (C type), 655  
`SUNDomEigEstimator_Ops_` (C struct), 655  
`SUNDomEigEstimator_Ops_` (class in *sundials4py.core*), 849  
`SUNDomEigEstimator_Ops_.destroy` (C member), 656  
`SUNDomEigEstimator_Ops_.estimate` (C member), 656  
`SUNDomEigEstimator_Ops_.getnumatimescalls` (C member), 656  
`SUNDomEigEstimator_Ops_.getnumiters` (C member), 656  
`SUNDomEigEstimator_Ops_.getres` (C member), 656  
`SUNDomEigEstimator_Ops_.initialize` (C member), 656  
`SUNDomEigEstimator_Ops_.setatimes` (C member), 655  
`SUNDomEigEstimator_Ops_.setinitialguess` (C member), 655  
`SUNDomEigEstimator_Ops_.setmaxiters` (C member), 655  
`SUNDomEigEstimator_Ops_.setnumpreprocessors` (C member), 655  
`SUNDomEigEstimator_Ops_.setreltol` (C member),



655			
SUNDomEigEstimator_Ops_.write (C member),	656		
SUNDomEigEstimator_Power (C function),	658		
SUNDomEigEstimator_SetATimes (C function),	652		
SUNDomEigEstimator_SetInitialGuess (C function),	653		
SUNDomEigEstimator_SetMaxIters (C function),	653		
SUNDomEigEstimator_SetNumPreprocessIters (C function),	652		
SUNDomEigEstimator_SetOptions (C function),	651		
SUNDomEigEstimator_SetRelTol (C function),	653		
SUNDomEigEstimator_Write (C function),	654		
SUNDomEigEstimatorContent_Power_ (class in sundials4py.core),	849		
SUNErrCode (C type),	54		
SUNErrCode (class in sundials4py.core),	849		
SUNErrorHandlerFn (C type),	55		
SUNFALSE (C macro),	47		
SUNFileClose (C function),	834		
SUNFileOpen (C function),	834		
SUNFullRhsMode (C enum),	687		
SUNFullRhsMode (class in sundials4py.core),	850		
SUNFullRhsMode.SUN_FULLLRHS_END (C enumerator),	687		
SUNFullRhsMode.SUN_FULLLRHS_OTHER (C enumerator),	687		
SUNFullRhsMode.SUN_FULLLRHS_START (C enumerator),	687		
SUNGetErrMsg (C function),	54		
SUNGramSchmidtType (class in sundials4py.core),	850		
SUNHipBlockReduceAtomicExecPolicy (C++ function),	482		
SUNHipBlockReduceExecPolicy (C++ function),	482		
SUNHipExecPolicy (C++ type),	481		
SUNHipGridStrideExecPolicy (C++ function),	482		
SUNHipThreadDirectExecPolicy (C++ function),	482		
sunindextype (C type),	47		
SUNLinearSolver (C type),	563		
SUNLINEARSOLVER_BAND (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_CUSOLVERS_BATCHQR (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_CUSTOM (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_DENSE (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_DIRECT (sundials4py.core.SUNLinearSolver_Type attribute),	851		
SUNLINEARSOLVER_GINKGO (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_GINKGOBATCH (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLinearSolver_ID (C enum),	567		
SUNLinearSolver_ID (class in sundials4py.core),	850		
SUNLINEARSOLVER_ITERATIVE (sundials4py.core.SUNLinearSolver_Type attribute),	851		
SUNLINEARSOLVER_KLU (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_KOKKOSDENSE (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_LAPACKBAND (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_LAPACKDENSE (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_MAGMADENSE (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_MATRIX_EMBEDDED (sundials4py.core.SUNLinearSolver_Type attribute),	851		
SUNLINEARSOLVER_MATRIX_ITERATIVE (sundials4py.core.SUNLinearSolver_Type attribute),	851		
SUNLINEARSOLVER_ONEMKLDENSE (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLinearSolver_Ops (C type),	564		
SUNLINEARSOLVER_PCG (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_SPBCGS (sundials4py.core.SUNLinearSolver_ID attribute),	850		
SUNLINEARSOLVER_SPGMR (sundials4py.core.SUNLinearSolver_ID attribute),	851		
SUNLINEARSOLVER_SPGMR (sundials4py.core.SUNLinearSolver_ID attribute),	851		
SUNLINEARSOLVER_SPTFQMR (sundials4py.core.SUNLinearSolver_ID attribute),	851		
SUNLINEARSOLVER_SUPERLUDIST (sundials4py.core.SUNLinearSolver_ID attribute),	851		
SUNLINEARSOLVER_SUPERLUMT (sundials4py.core.SUNLinearSolver_ID attribute),	851		

- als4py.core.SUNLinearSolver\_ID* attribute), 851
- SUNLinearSolver\_Type* (C enum), 556
- SUNLinearSolver\_Type* (class in *sundials4py.core*), 851
- SUNLinearSolver\_Type.SUNLINEARSOLVER\_DIRECT* (C enumerator), 556
- SUNLinearSolver\_Type.SUNLINEARSOLVER\_ITERATIVE* (C enumerator), 557
- SUNLinearSolver\_Type.SUNLINEARSOLVER\_MATRIX\_EMBEDDED* (C enumerator), 557
- SUNLinearSolver\_Type.SUNLINEARSOLVER\_MATRIX\_ITERATIVE* (C enumerator), 557
- SUNLinSol\_Band* (C function), 572
- SUNLinSol\_cuSolverSp\_batchQR* (C function), 609
- SUNLinSol\_cuSolverSp\_batchQR\_GetDescription* (C function), 610
- SUNLinSol\_cuSolverSp\_batchQR\_GetDeviceSpace* (C function), 610
- SUNLinSol\_cuSolverSp\_batchQR\_SetDescription* (C function), 610
- SUNLinSol\_Dense* (C function), 574
- SUNLinSol\_KLU* (C function), 575
- SUNLinSol\_KLUGetCommon* (C function), 576
- SUNLinSol\_KLUGetCommon.sun\_klu\_common* (C type), 577
- SUNLinSol\_KLUGetNumeric* (C function), 576
- SUNLinSol\_KLUGetNumeric.sun\_klu\_numeric* (C type), 576
- SUNLinSol\_KLUGetSymbolic* (C function), 576
- SUNLinSol\_KLUGetSymbolic.sun\_klu\_symbolic* (C type), 576
- SUNLinSol\_KLUREInit* (C function), 575
- SUNLinSol\_KLUSetOrdering* (C function), 576
- SUNLinSol\_LapackBand* (C function), 578
- SUNLinSol\_LapackDense* (C function), 580
- SUNLinSol\_MagmaDense* (C function), 582
- SUNLinSol\_MagmaDense\_SetAsync* (C function), 582
- SUNLinSol\_OneMklDense* (C function), 584
- SUNLinSol\_PCG* (C function), 585
- SUNLinSol\_PCGSetMaxl* (C function), 586
- SUNLinSol\_PCGSetPrecType* (C function), 586
- SUNLinSol\_SPBCGS* (C function), 588
- SUNLinSol\_SPBCGSSetMaxl* (C function), 589
- SUNLinSol\_SPBCGSSetPrecType* (C function), 589
- SUNLinSol\_SPGMR* (C function), 592
- SUNLinSol\_SPGMRSetGSType* (C function), 593
- SUNLinSol\_SPGMRSetMaxRestarts* (C function), 593
- SUNLinSol\_SPGMRSetPrecType* (C function), 592
- SUNLinSol\_SPGMR* (C function), 596
- SUNLinSol\_SPGMRSetGSType* (C function), 596
- SUNLinSol\_SPGMRSetMaxRestarts* (C function), 597
- SUNLinSol\_SPGMRSetPrecType* (C function), 596
- SUNLinSol\_SPTFQMR* (C function), 600
- SUNLinSol\_SPTFQMRSetMaxl* (C function), 601
- SUNLinSol\_SPTFQMRSetPrecType* (C function), 600
- SUNLinSol\_SuperLUDIST* (C function), 603
- SUNLinSol\_SuperLUDIST\_GetBerr* (C function), 604
- SUNLinSol\_SuperLUDIST\_GetGridinfo* (C function), 604
- SUNLinSol\_SuperLUDIST\_GetLUstruct* (C function), 604
- SUNLinSol\_SuperLUDIST\_GetScalePermstruct* (C function), 604
- SUNLinSol\_SuperLUDIST\_GetSOLVEstruct* (C function), 604
- SUNLinSol\_SuperLUDIST\_GetSuperLUOptions* (C function), 604
- SUNLinSol\_SuperLUDIST\_GetSuperLUStat* (C function), 604
- SUNLinSol\_SuperLUMT* (C function), 606
- SUNLinSol\_SuperLUMTSetOrdering* (C function), 606
- SUNLinSolFree* (C function), 558
- SUNLinSolFreeEmpty* (C function), 566
- SUNLinSolGetID* (C function), 557
- SUNLinSolGetType* (C function), 557
- SUNLinSolInitialize* (C function), 557
- SUNLinSolLastFlag* (C function), 561
- SUNLinSolNewEmpty* (C function), 566
- SUNLinSolNumIters* (C function), 561
- SUNLinSolNumIters\_GinkgoBatch* (C function), 615
- SUNLinSolResid* (C function), 561
- SUNLinSolResNorm* (C function), 561
- SUNLinSolSetATimes* (C function), 560
- SUNLinSolSetOptions* (C function), 559
- SUNLinSolSetPreconditioner* (C function), 560
- SUNLinSolSetScalingVectors* (C function), 560
- SUNLinSolSetup* (C function), 557
- SUNLinSolSetZeroGuess* (C function), 560
- SUNLinSolSolve* (C function), 558
- SUNLinSolSpace* (C function), 561
- SUNLogErrorHandlerFn* (C function), 55
- SUNLogger* (C type), 60
- SUNLogger\_* (class in *sundials4py.core*), 851
- SUNLogger\_Create* (C function), 60
- SUNLogger\_CreateFromEnv* (C function), 61
- SUNLogger\_Destroy* (C function), 64
- SUNLogger\_Flush* (C function), 63
- SUNLogger\_GetOutputRank* (C function), 63
- SUNLogger\_QueueMsg* (C function), 63
- SUNLogger\_SetDebugFile* (C function), 63
- SUNLogger\_SetDebugFilename* (C function), 62
- SUNLogger\_SetErrorFile* (C function), 61
- SUNLogger\_SetErrorFilename* (C function), 61
- SUNLogger\_SetInfoFile* (C function), 62
- SUNLogger\_SetInfoFilename* (C function), 62
- SUNLogger\_SetWarningFile* (C function), 62
- SUNLogger\_SetWarningFilename* (C function), 61

- SUNLogLevel (C enum), 60
- SUNLogLevel (class in *sundials4py.core*), 851
- SUNLogLevel.SUN\_LOGLEVEL\_ALL (C enumerator), 60
- SUNLogLevel.SUN\_LOGLEVEL\_DEBUG (C enumerator), 60
- SUNLogLevel.SUN\_LOGLEVEL\_ERROR (C enumerator), 60
- SUNLogLevel.SUN\_LOGLEVEL\_INFO (C enumerator), 60
- SUNLogLevel.SUN\_LOGLEVEL\_NONE (C enumerator), 60
- SUNLogLevel.SUN\_LOGLEVEL\_WARNING (C enumerator), 60
- SUNMatClone (C function), 514
- SUNMatCopy (C function), 515
- SUNMatCopyOps (C function), 513
- SUNMatDestroy (C function), 514
- SUNMatFreeEmpty (C function), 513
- SUNMatGetID (C function), 514
- SUNMatHermitianTransposeVec (C function), 516
- SUNMatMatvec (C function), 515
- SUNMatMatvecSetup (C function), 515
- SUNMatNewEmpty (C function), 513
- SUNMatrix (C type), 511
- SUNMATRIX\_BAND (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMATRIX\_CUSPARSE (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMatrix\_cuSparse\_BlockColumns (C function), 536
- SUNMatrix\_cuSparse\_BlockData (C function), 536
- SUNMatrix\_cuSparse\_BlockNNZ (C function), 536
- SUNMatrix\_cuSparse\_BlockRows (C function), 536
- SUNMatrix\_cuSparse\_Columns (C function), 536
- SUNMatrix\_cuSparse\_CopyFromDevice (C function), 536
- SUNMatrix\_cuSparse\_CopyToDevice (C function), 536
- SUNMatrix\_cuSparse\_Data (C function), 536
- SUNMatrix\_cuSparse\_IndexPointers (C function), 536
- SUNMatrix\_cuSparse\_IndexValues (C function), 536
- SUNMatrix\_cuSparse\_MakeCSR (C function), 535
- SUNMatrix\_cuSparse\_MatDescr (C function), 536
- SUNMatrix\_cuSparse\_NewBlockCSR (C function), 535
- SUNMatrix\_cuSparse\_NewCSR (C function), 535
- SUNMatrix\_cuSparse\_NNZ (C function), 536
- SUNMatrix\_cuSparse\_NumBlocks (C function), 536
- SUNMatrix\_cuSparse\_Rows (C function), 535
- SUNMatrix\_cuSparse\_SetFixedPattern (C function), 537
- SUNMatrix\_cuSparse\_SetKernelExecPolicy (C function), 537
- SUNMatrix\_cuSparse\_SparseType (C function), 536
- SUNMATRIX\_CUSTOM (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMATRIX\_DENSE (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMATRIX\_GINKGO (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMATRIX\_GINKGOBATCH (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMatrix\_ID (C type), 513
- SUNMatrix\_ID (class in *sundials4py.core*), 851
- SUNMATRIX\_KOKKOSDENSE (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMatrix\_MagmaDense (C function), 520
- SUNMATRIX\_MAGMADENSE (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMatrix\_MagmaDense\_Block (C function), 522
- SUNMatrix\_MagmaDense\_BlockColumn (C function), 522
- SUNMatrix\_MagmaDense\_BlockColumns (C function), 521
- SUNMatrix\_MagmaDense\_BlockData (C function), 521
- SUNMatrix\_MagmaDense\_BlockRows (C function), 521
- SUNMatrix\_MagmaDense\_Column (C function), 522
- SUNMatrix\_MagmaDense\_Columns (C function), 520
- SUNMatrix\_MagmaDense\_CopyFromDevice (C function), 523
- SUNMatrix\_MagmaDense\_CopyToDevice (C function), 523
- SUNMatrix\_MagmaDense\_Data (C function), 521
- SUNMatrix\_MagmaDense\_LData (C function), 521
- SUNMatrix\_MagmaDense\_NumBlocks (C function), 521
- SUNMatrix\_MagmaDense\_Rows (C function), 520
- SUNMatrix\_MagmaDenseBlock (C function), 520
- SUNMatrix\_OneMklDense (C++ function), 524
- SUNMATRIX\_ONEMKLDENSE (*sundials4py.core.SUNMatrix\_ID* attribute), 851
- SUNMatrix\_OneMklDense\_Block (C function), 527
- SUNMatrix\_OneMklDense\_BlockColumn (C function), 527
- SUNMatrix\_OneMklDense\_BlockColumns (C function), 526
- SUNMatrix\_OneMklDense\_BlockData (C function), 527
- SUNMatrix\_OneMklDense\_BlockLData (C function), 527
- SUNMatrix\_OneMklDense\_BlockRows (C function), 526
- SUNMatrix\_OneMklDense\_Column (C function), 526
- SUNMatrix\_OneMklDense\_Columns (C function), 525
- SUNMatrix\_OneMklDense\_CopyFromDevice (C function), 528
- SUNMatrix\_OneMklDense\_CopyToDevice (C function), 528
- SUNMatrix\_OneMklDense\_Data (C function), 526

- SUNMatrix\_OneMklDense\_LData (C function), 526  
 SUNMatrix\_OneMklDense\_NumBlocks (C function), 526  
 SUNMatrix\_OneMklDense\_Rows (C function), 525  
 SUNMatrix\_OneMklDenseBlock (C++ function), 525  
 SUNMatrix\_SLUNRloc (C function), 544  
 SUNMATRIX\_SLUNRLOC (sundials4py.core.SUNMatrix\_ID attribute), 851  
 SUNMatrix\_SLUNRloc\_OwnData (C function), 544  
 SUNMatrix\_SLUNRloc\_Print (C function), 544  
 SUNMatrix\_SLUNRloc\_ProcessGrid (C function), 544  
 SUNMatrix\_SLUNRloc\_SuperMatrix (C function), 544  
 SUNMATRIX\_SPARSE (sundials4py.core.SUNMatrix\_ID attribute), 851  
 SUNMatScaleAdd (C function), 515  
 SUNMatScaleAddI (C function), 515  
 SUNMatSpace (C function), 514  
 SUNMatZero (C function), 515  
 SUNMemory (C type), 705  
 SUNMemory.SUNMemory\_ (C struct), 705  
 SUNMemory.SUNMemory\_.bytes (C member), 705  
 SUNMemory.SUNMemory\_.own (C member), 705  
 SUNMemory.SUNMemory\_.ptr (C member), 705  
 SUNMemory.SUNMemory\_.stride (C member), 705  
 SUNMemory.SUNMemory\_.type (C member), 705  
 SUNMemoryHelper (C type), 706  
 SUNMemoryHelper.SUNMemoryHelper\_ (C struct), 706  
 SUNMemoryHelper.SUNMemoryHelper\_.content (C member), 706  
 SUNMemoryHelper.SUNMemoryHelper\_.ops (C member), 706  
 SUNMemoryHelper.SUNMemoryHelper\_.queue (C member), 706  
 SUNMemoryHelper.SUNMemoryHelper\_.sunctx (C member), 706  
 SUNMemoryHelper\_ (class in sundials4py.core), 851  
 SUNMemoryHelper\_Alias (C function), 708  
 SUNMemoryHelper\_Alloc (C function), 707  
 SUNMemoryHelper\_Alloc\_Cuda (C function), 712  
 SUNMemoryHelper\_Alloc\_Hip (C function), 715  
 SUNMemoryHelper\_Alloc\_Sycl (C function), 717  
 SUNMemoryHelper\_AllocStrided (C function), 707  
 SUNMemoryHelper\_AllocStrided\_Cuda (C function), 712  
 SUNMemoryHelper\_AllocStrided\_Hip (C function), 715  
 SUNMemoryHelper\_AllocStrided\_Sycl (C function), 717  
 SUNMemoryHelper\_Clone (C function), 710  
 SUNMemoryHelper\_Copy (C function), 708  
 SUNMemoryHelper\_Copy\_Cuda (C function), 713  
 SUNMemoryHelper\_Copy\_Hip (C function), 715  
 SUNMemoryHelper\_Copy\_Sycl (C function), 718  
 SUNMemoryHelper\_CopyAsync (C function), 710  
 SUNMemoryHelper\_CopyAsync\_Cuda (C function), 713  
 SUNMemoryHelper\_CopyAsync\_Hip (C function), 716  
 SUNMemoryHelper\_CopyAsync\_Sycl (C function), 718  
 SUNMemoryHelper\_CopyOps (C function), 709  
 SUNMemoryHelper\_Cuda (C function), 712  
 SUNMemoryHelper\_Dealloc (C function), 708  
 SUNMemoryHelper\_Dealloc\_Cuda (C function), 713  
 SUNMemoryHelper\_Dealloc\_Hip (C function), 715  
 SUNMemoryHelper\_Dealloc\_Sycl (C function), 718  
 SUNMemoryHelper\_Destroy (C function), 711  
 SUNMemoryHelper\_GetAllocStats (C function), 709  
 SUNMemoryHelper\_GetAllocStats\_Cuda (C function), 714  
 SUNMemoryHelper\_GetAllocStats\_Hip (C function), 716  
 SUNMemoryHelper\_GetAllocStats\_Sycl (C function), 719  
 SUNMemoryHelper\_Hip (C function), 714  
 SUNMemoryHelper\_NewEmpty (C function), 709  
 SUNMemoryHelper\_Ops (C type), 706  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_ (C struct), 706  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.alloc (C member), 706  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.allocstrided (C member), 706  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.clone (C member), 707  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.copy (C member), 706  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.copyasync (C member), 706  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.dealloc (C member), 706  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.destroy (C member), 707  
 SUNMemoryHelper\_Ops.SUNMemoryHelper\_Ops\_.getallocstats (C member), 707  
 SUNMemoryHelper\_Ops\_ (class in sundials4py.core), 852  
 SUNMemoryHelper\_SetDefaultQueue (C function), 710  
 SUNMemoryHelper\_Sycl (C function), 717  
 SUNMemoryHelper\_Sys (C function), 711  
 SUNMemoryHelper\_Wrap (C function), 708  
 SUNMemoryNewEmpty (C function), 705  
 SUNMemoryType (C enum), 706  
 SUNMemoryType (class in sundials4py.core), 852  
 SUNMemoryType.SUNMEMTYPE\_DEVICE (C enumerator), 706  
 SUNMemoryType.SUNMEMTYPE\_HOST (C enumerator), 706  
 SUNMemoryType.SUNMEMTYPE\_PINNED (C enumerator), 706



- SUNMemoryType.SUNMEMTYPE\_UVM (*C enumerator*), 706  
 SUNMEMTYPE\_DEVICE (*sundials4py.core.SUNMemoryType attribute*), 852  
 SUNMEMTYPE\_HOST (*sundials4py.core.SUNMemoryType attribute*), 852  
 SUNMEMTYPE\_PINNED (*sundials4py.core.SUNMemoryType attribute*), 852  
 SUNMEMTYPE\_UVM (*sundials4py.core.SUNMemoryType attribute*), 852  
 SUNMPIAbortErrHandlerFn (*C function*), 56  
 SUNNonlinearSolver (*C type*), 631  
 SUNNONLINEARSOLVER\_FIXEDPOINT (*sundials4py.core.SUNNonlinearSolver\_Type attribute*), 852  
 SUNNonlinearSolver\_Ops (*C type*), 631  
 SUNNONLINEARSOLVER\_ROOTFIND (*sundials4py.core.SUNNonlinearSolver\_Type attribute*), 852  
 SUNNonlinearSolver\_Type (*C enum*), 623  
 SUNNonlinearSolver\_Type (*class in sundials4py.core*), 852  
 SUNNonlinearSolver\_-  
     Type.SUNNONLINEARSOLVER\_FIXEDPOINT (*C enumerator*), 624  
 SUNNonlinearSolver\_-  
     Type.SUNNONLINEARSOLVER\_ROOTFIND (*C enumerator*), 624  
 SUNNonlinSol\_FixedPoint (*C function*), 642  
 SUNNonlinSol\_Newton (*C function*), 640  
 SUNNonlinSol\_PetscSNES (*C function*), 645  
 SUNNonlinSolConvTestFn (*C type*), 630  
 SUNNonlinSolFree (*C function*), 625  
 SUNNonlinSolFreeEmpty (*C function*), 633  
 SUNNonlinSolGetCurIter (*C function*), 628  
 SUNNonlinSolGetNumConvFails (*C function*), 628  
 SUNNonlinSolGetNumIters (*C function*), 628  
 SUNNonlinSolGetPetscError\_PetscSNES (*C function*), 646  
 SUNNonlinSolGetSNES\_PetscSNES (*C function*), 646  
 SUNNonlinSolGetSysFn\_FixedPoint (*C function*), 642  
 SUNNonlinSolGetSysFn\_Newton (*C function*), 640  
 SUNNonlinSolGetSysFn\_PetscSNES (*C function*), 646  
 SUNNonlinSolGetType (*C function*), 624  
 SUNNonlinSolInitialize (*C function*), 624  
 SUNNonlinSolLSetupFn (*C type*), 629  
 SUNNonlinSolLSolveFn (*C type*), 629  
 SUNNonlinSolNewEmpty (*C function*), 632  
 SUNNonlinSolSetConvTestFn (*C function*), 627  
 SUNNonlinSolSetDamping\_FixedPoint (*C function*), 643  
 SUNNonlinSolSetLSetupFn (*C function*), 626  
 SUNNonlinSolSetLSolveFn (*C function*), 627  
 SUNNonlinSolSetMaxIters (*C function*), 627  
 SUNNonlinSolSetOptions (*C function*), 625  
 SUNNonlinSolSetSysFn (*C function*), 626  
 SUNNonlinSolSetup (*C function*), 624  
 SUNNonlinSolSolve (*C function*), 624  
 SUNNonlinSolSysFn (*C type*), 628  
 SUNOutputFormat (*C enum*), 47  
 SUNOutputFormat (*class in sundials4py.core*), 852  
 SUNPrecType (*class in sundials4py.core*), 852  
 SUNProfiler (*C type*), 65  
 SUNProfiler\_ (*class in sundials4py.core*), 852  
 SUNProfiler\_Begin (*C function*), 65  
 SUNProfiler\_Create (*C function*), 65  
 SUNProfiler\_End (*C function*), 66  
 SUNProfiler\_Free (*C function*), 65  
 SUNProfiler\_GetElapsedTime (*C function*), 66  
 SUNProfiler\_GetTimerResolution (*C function*), 66  
 SUNProfiler\_Print (*C function*), 66  
 SUNProfiler\_Reset (*C function*), 66  
 SUNPSolveFn (*C type*), 562  
 SUNPSolveFn (*C type*), 562  
 sunrealtype (*C type*), 46  
 SUNSparseFromBandMatrix (*C function*), 542  
 SUNSparseFromDenseMatrix (*C function*), 542  
 SUNSparseMatrix (*C function*), 542  
 SUNSparseMatrix\_Columns (*C function*), 543  
 SUNSparseMatrix\_Data (*C function*), 543  
 SUNSparseMatrix\_IndexPointers (*C function*), 543  
 SUNSparseMatrix\_IndexValues (*C function*), 543  
 SUNSparseMatrix\_NNZ (*C function*), 543  
 SUNSparseMatrix\_NP (*C function*), 543  
 SUNSparseMatrix\_Print (*C function*), 542  
 SUNSparseMatrix\_Realloc (*C function*), 542  
 SUNSparseMatrix\_Reallocate (*C function*), 542  
 SUNSparseMatrix\_Rows (*C function*), 543  
 SUNSparseMatrix\_SparseType (*C function*), 543  
 SUNStepper (*C type*), 683  
 SUNStepper\_ (*class in sundials4py.core*), 852  
 SUNStepper\_Create (*C function*), 684  
 SUNStepper\_Destroy (*C function*), 684  
 SUNStepper\_Evolve (*C function*), 684  
 SUNStepper\_FullRhs (*C function*), 685  
 SUNStepper\_GetContent (*C function*), 687  
 SUNStepper\_GetLastFlag (*C function*), 688  
 SUNStepper\_GetNumSteps (*C function*), 687  
 SUNStepper\_OneStep (*C function*), 685  
 SUNStepper\_ReInit (*C function*), 685  
 SUNStepper\_Reset (*C function*), 685  
 SUNStepper\_ResetCheckpointIndex (*C function*), 686  
 SUNStepper\_SetContent (*C function*), 687  
 SUNStepper\_SetDestroyFn (*C function*), 690  
 SUNStepper\_SetEvolveFn (*C function*), 688

SUNStepper\_SetForcing (*C function*), 686  
SUNStepper\_SetForcingFn (*C function*), 690  
SUNStepper\_SetFullRhsFn (*C function*), 689  
SUNStepper\_SetGetNumStepsFn (*C function*), 690  
SUNStepper\_SetLastFlag (*C function*), 688  
SUNStepper\_SetOneStepFn (*C function*), 688  
SUNStepper\_SetReInitFn (*C function*), 689  
SUNStepper\_SetResetCheckpointIndexFn (*C function*), 689  
SUNStepper\_SetResetFn (*C function*), 689  
SUNStepper\_SetStepDirection (*C function*), 686  
SUNStepper\_SetStepDirectionFn (*C function*), 690  
SUNStepper\_SetStopTime (*C function*), 686  
SUNStepper\_SetStopTimeFn (*C function*), 689  
SUNStepperDestroyFn (*C type*), 691  
SUNStepperEvolveFn (*C type*), 690  
SUNStepperFullRhsFn (*C type*), 691  
SUNStepperGetNumStepsFn (*C type*), 691  
SUNStepperOneStepFn (*C type*), 690  
SUNStepperReInitFn (*C type*), 691  
SUNStepperResetCheckpointIndexFn (*C type*), 691  
SUNStepperResetFn (*C type*), 691  
SUNStepperSetForcingFn (*C type*), 691  
SUNStepperSetStepDirectionFn (*C type*), 691  
SUNStepperSetStopTimeFn (*C type*), 691  
SUNSYCLBlockReduceExecPolicy (*C++ function*), 487  
SUNSYCLExecPolicy (*C++ type*), 486  
SUNSYCLGridStrideExecPolicy (*C++ function*), 487  
SUNSYCLThreadDirectExecPolicy (*C++ function*), 487  
SUNTRUE (*C macro*), 47

## U

User main program, 73

## V

Vector (*C++ class*), 492  
Vector::~~Vector (*C++ function*), 493  
Vector::exec\_space (*C++ type*), 492  
Vector::get (*C++ function*), 493  
Vector::host\_view\_type (*C++ type*), 492  
Vector::HostView (*C++ function*), 493  
Vector::Length (*C++ function*), 493  
Vector::memory\_space (*C++ type*), 492  
Vector::operator N\_Vector (*C++ function*), 493  
Vector::operator= (*C++ function*), 493  
Vector::range\_policy (*C++ type*), 492  
Vector::size\_type (*C++ type*), 492  
Vector::Vector (*C++ function*), 492  
Vector::View (*C++ function*), 493  
Vector::view\_type (*C++ type*), 492  
vector\_type (*C++ type*), 497

## W

weighted root-mean-square norm, 22