ARKode Documentation

Release 1.0

Daniel R. Reynolds

November 18, 2013

CONTENTS

This is the documentation for ARKode, an adaptive step time integration package for stiff, nonstiff and multi-rate systems of ordinary differential equations (ODEs). The ARKode solver is a component of the SUNDIALS suite of nonlinear and differential/algebraic equation solvers. It is designed to have a similar user experience to the CVODE solver, including user modes to allow adaptive integration to specified output times, return after each internal step and root-finding capabilities, and for calculations both in serial and parallel (via MPI). The default integration and solver options should apply to most users, though complete control over all internal parameters and time adaptivity algorithms is enabled through optional interface routines.

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

Due to its similarities in both function and design with the CVODE package, a significant portion of this documentation has been directly adapted from the CVODE documentation [HS2012].

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

CHAPTER

ONE

INTRODUCTION

The ARKode solver library provides an adaptive-step time integration package for stiff, nonstiff and multi-rate systems of ordinary differential equations (ODEs) given in explicit form

$$M\dot{y} = f_E(t,y) + f_I(t,y), \qquad y(t_0) = y_0, \tag{1.1}$$

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

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

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

The methods used in ARKode are adaptive-step additive Runge Kutta methods. Such methods are defined through combining two complementary Runge-Kutta methods: one explicit (ERK) and the other diagonally implicit (DIRK). Through appropriately partitioning the ODE system into explicit and implicit components (1.1), such methods have the potential to enable accurate and efficient time integration of multi-rate systems of ordinary differential equations. A key feature allowing for high efficiency of these methods is that only the components in $f_I(t,y)$ must be solved implicitly, allowing for splittings tuned for use with optimal implicit solvers.

This framework allows for significant freedom over the constitutive methods used for each component, and ARKode is packaged with a wide array of built-in methods for use. These built-in Butcher tables include adaptive explicit methods of orders 2-6, adaptive implicit methods of orders 2-5, and adaptive IMEX methods of orders 3-5.

For problems that include nonzero implicit term $f_I(t,y)$, the resulting implicit system (assumed nonlinear) is solved approximately at each integration step, using a modified Newton method, an Inexact Newton method, or an accelerated fixed-point solver. For implicit problems using a Newton-based solver and the serial NVECTOR module in SUNDIALS, ARKode provides both direct (dense and band) and preconditioned Krylov iterative (GMRES, BiCGStab, TFQMR, FGMRES, PCG) linear solvers. When used with the parallel NVECTOR module or a user-provided vector data structure, only the Krylov solvers are available, although a user may supply their own linear solver for any data structures if desired.

The guide is separated into sections focused on the major aspects of the ARKode library. In the next section we provide a thorough presentation of the underlying *mathematics* that relate these algorithms together. We follow this with overview of how the source code for ARKode is *organized*. The largest section follows, providing a full account of the ARKode user interface, including a description of all user-accessible functions and outlines for ARKode usage for serial and parallel applications. Since ARKode is written in C, we first present *the C and C++ interface*, followed with a separate section on *using ARKode within Fortran applications*. The following three sections discuss shared features between ARKode and the rest of the SUNDIALS library: *vector data structures*, *linear solvers*, 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.

CHAPTER

TWO

MATHEMATICAL CONSIDERATIONS

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

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

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

M is a user-specified nonsingular operator from $\mathbb{R}^N \to \mathbb{R}^N$. This operator may depend on t but is currently assumed to be independent of y. For standard systems of ordinary differential equations and for problems arising from the spatial semi-discretization of partial differential equations using finite difference or finite volume methods, M is typically the identity matrix, I. For PDEs using a finite-element spatial semi-discretization M is typically a well-conditioned mass matrix.

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

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

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

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

2.1 Additive Runge-Kutta methods

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

$$
\begin{aligned}
Mz_i &= My_{n-1} + h_n \sum_{j=0}^{i-1} A^E_{i,j} f_E(t_{n,j}, z_j) + h_n \sum_{j=0}^{i} A^I_{i,j} f_I(t_{n,j}, z_j), \quad i = 1, \ldots, s, \\
My_n &= My_{n-1} + h_n \sum_{i=0}^{s} b_i \left(f_E(t_{n,i}, z_i) + f_I(t_{n,i}, z_i) \right), \\
M\tilde{y}_n &= My_{n-1} + h_n \sum_{i=0}^{s} \tilde{b}_i \left(f_E(t_{n,i}, z_i) + f_I(t_{n,i}, z_i) \right).
\end{aligned} \tag{2.2}
$$

Here the y_n are computed approximations to $y(t_n)$, $\tilde{y}_n$ are lower-order embedded solutions (used in error estimation), and $h_n \equiv t_n - t_{n-1}$ is the step size. The internal stage times are abbreviated using the notation $t_{n,j} = t_{n-1} + c_j h_n$. The ARK method is primarily defined through the coefficients $A^E \in \mathbb{R}^{s \times s}$, $A^I \in \mathbb{R}^{s \times s}$, $b \in \mathbb{R}^s$ and $c \in \mathbb{R}^s$, that correspond with the explicit and implicit Butcher tables. We note that ARKode enforces the constraint that these tables must share b and c between the explicit and implicit methods in an ARK pair.

The user of ARKode must choose appropriately between one of three classes of methods: *multi-rate*, *nonstiff* and *stiff*. All of ARKode's available Butcher tables encoding the coefficients c, A^E, A^I, b and $\tilde{b}$ are further described in the *Appendix: Butcher tables*.

For multi-rate problems, a user should provide both of the functions f_E and f_I that define the IVP system. For such problems, ARKode currently implements the ARK methods proposed in [KC2003], allowing for methods having order $q = \{3, 4, 5\}$. The tables for these methods are given in the section *Additive Butcher tables*.

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

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

In this scenario, the Butcher table $A^I = 0$ in (2.2), and the ARK methods reduce to classical explicit Runge-Kutta methods (ERK). For these classes of methods, ARKode allows orders of accuracy $q = \{2, 3, 4, 5, 6\}$, with embeddings of orders $p = \{1, 2, 3, 4, 5\}$. These default to the *Heun-Euler-2-1-2*, *Bogacki-Shampine-4-2-3*, *Zonneveld-5-3-4*, *Cash-Karp-6-4-5* and *Verner-8-5-6* methods, respectively.

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

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

Similarly to ERK methods, in this scenario the Butcher table $A^E = 0$ in (2.2), and the ARK methods reduce to classical diagonally-implicit Runge-Kutta methods (DIRK). For these classes of methods, ARKode allows orders of accuracy $q = \{2, 3, 4, 5\}$, with embeddings of orders $p = \{1, 2, 3, 4\}$. These default to the *SDIRK-2-1-2*, *ARK-4-2-3 (implicit)*, *SDIRK-5-3-4* and *ARK-8-4-5 (implicit)* methods, respectively.

2.2 Nonlinear solver methods

For both the DIRK and ARK methods corresponding to (2.1) and (2.4), a nonlinear system

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

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

$$a_i \equiv My_{n-1} + h_n \sum_{j=0}^{i-1} \left[A^E_{i,j} f_E(t_{n,j}, z_j) + A^I_{i,j} f_I(t_{n,j}, z_j)\right]$$

for the ARK methods, or

$$a_i \equiv My_{n-1} + h_n \sum_{j=0}^{i-1} A^I_{i,j} f_I(t_{n,j}, z_j)$$

for the DIRK methods. For these nonlinear systems, ARKode allows a choice of solution strategy.

The default solver choice is a variant of Newton's method,

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

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

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

in which

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

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

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

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

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

While ARKode uses the Newton iteration as its default solver due to its increased robustness on very stiff problems, it is highly recommended that users also consider the fixed-point solver for their when attempting a new problem.

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

2.3 Linear solver methods

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

- dense direct solvers, using either an internal SUNDIALS implementation or a BLAS/LAPACK implementation (serial version only),
- band direct solvers, using either an internal SUNDIALS implementation or a BLAS/LAPACK implementation (serial version only),
- SPGMR, a scaled, preconditioned GMRES (Generalized Minimal Residual) solver without restarts,
- SPBCG, a scaled, preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable) solver,
- SPTFQMR, a scaled, preconditioned TFQMR (Transpose-free Quasi-Minimal Residual) solver,
- SPFGMR, a scaled, preconditioned Flexible GMRES (Generalized Minimal Residual) solver without restarts, or
- PCG, a preconditioned conjugate gradient solver for symmetric linear systems.

For large stiff systems where direct methods are infeasible, the combination of an implicit integrator and a preconditioned Krylov method (SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG) can yield a powerful tool because it combines established methods for stiff integration, nonlinear solver iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant sources of stiffness, in the form of a user-supplied preconditioner matrix [BH1989]. We note that the direct linear solvers provided by SUNDIALS (dense and band), as well as the direct linear solvers accessible through LAPACK, can only be used with the serial vector representations.

In the case that a direct linear solver is used (dense or band), ARKode utilizes a *modified Newton iteration*. In such methods, the matrix $\mathcal{A}$ is held fixed for multiple Newton iterations. More precisely, each Newton iteration is computed from the modified equation

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

in which

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

Here, the solution $\tilde{y}$ and step size $\tilde{h}$ upon which the modified Jacobian rely, are merely values of the solution and step size from a previous iteration. In other words, the matrix $\tilde{\mathcal{A}}$ is only computed rarely, and reused for repeated stage solves.

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

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

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

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

Here U is the unit roundoff, σ_0 is a dimensionless value, and w_j is the error weight defined in (2.13). In the dense case, this approach requires N evaluations of f_I, one for each column of J. In the band case, the columns of J are computed in groups, using the Curtis-Powell-Reid algorithm, with the number of f_I evaluations equal to the bandwidth.

In the case that an iterative linear solver is chosen, ARKode utilizes a Newton method variant called an *Inexact Newton iteration*. Here, the matrix $\mathcal{A}$ is not itself constructed since the algorithms only require the product of this matrix with a given vector. Additionally, each Newton system (2.7) is not solved completely, since these linear solvers are iterative (hence the "inexact" in the name). Resultingly. for these linear solvers $\mathcal{A}$ is applied in a matrix-free manner,

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

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

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

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

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

More specifically, in both of the Newton-based solvers, we update the Newton matrix $\tilde{\mathcal{A}}$ or preconditioner matrix P only in the following circumstances:

- when starting the problem,
- when more than 20 steps have been taken since the last update (this value may be changed via the *msbp* argument to `ARKodeSetMaxStepsBetweenLSet()`) or the *LSETUP_MSBP* argument to `FARKSETIIN()`,
- when the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma} - 1| > 0.2$ (this tolerance may be changed via the *dgmax* argument to `ARKodeSetDeltaGammaMax()`) or the *LSETUP_DGMAX* argument to `FARKSETRIN()`,
- when a non-fatal convergence failure just occurred, or
- when an error test failure just occurred.

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

- starting the problem,
- more than 50 steps have been taken since the last evaluation,
- a convergence failure occurred with an outdated matrix, and the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma} - 1| > 0.2$,
- a convergence failure occurred that forced a step size reduction.

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

2.4 Iteration Error Control

2.4.1 Choice of norm

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

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

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

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

Since $1/w_i$ represents a tolerance in the component y_i, a vector whose WRMS norm is 1 is regarded as "small." For brevity, we will typically drop the subscript WRMS on norms in the remainder of this section.

Additionally, for problems involving a non-identity mass matrix, $M \neq I$, the units of equation (2.1) may differ from the units of the solution y. In this case, ARKode may also construct a residual weight vector,

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

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

2.4.2 Nonlinear iteration error control

The stopping test for all of ARKode's nonlinear solvers is related to the subsequent local error test, with the goal of keeping the nonlinear iteration errors from interfering with local error control. Denoting the final computed value of each stage solution as $z_i^{(m)}$, and the true stage solution solving (2.5) as z_i, we want to ensure that the iteration error $z_i - z_i^{(m)}$ is "small" (recall that a norm less than 1 is already considered "small").

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

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

where the factor 0.3 is user-modifiable as the *crdown* input to the the function `ARKodeSetNonlinCRDown()` or the *NEWT_CRDOWN* argument to `FARKSETRIN()`.

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

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

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

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

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

2.4.3 Linear iteration error control

When a Krylov method is used to solve the linear systems (2.7), its errors must also be controlled. To this end, we approximate the linear iteration error in the solution vector $\delta^{(m)}$ using the preconditioned residual vector, e.g. $r = P\mathcal{A}\delta^{(m)} + PG$ for the case of left preconditioning (the role of the preconditioner is further elaborated on in the next section). In an attempt to ensure that the linear iteration errors do not interfere with the nonlinear solution error and local time integration error controls, we require that the norm of the preconditioned linear residual satisfies

$$\|r\| \le 0.05\epsilon.$$

Here ϵ is the same value as that used above for the nonlinear error control. The value 0.05 may be modified by the user through the `ARKSpilsSetEpsLin()` function; it cannot currently be modified from Fortran applications.

2.5 Preconditioning

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

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

The Krylov method is then applied to a system with the matrix $P^{-1}\mathcal{A}$, $\mathcal{A}P^{-1}$, or $P_L^{-1}\mathcal{A}P_R^{-1}$, instead of $\mathcal{A}$. In order to improve the convergence of the Krylov iteration, the preconditioner matrix P, or the product P_LP_R in the third case, should in some sense approximate the system matrix $\mathcal{A}$. Yet at the same time, in order to be cost-effective the matrix P

(or matrices P_L and P_R) should be reasonably efficient to evaluate and solve. Finding an optimal point in this tradeoff between rapid convergence and low cost can be quite challenging. Good choices are often problem-dependent (for example, see [BH1989] for an extensive study of preconditioners for reaction-transport systems).

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

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

2.6 Implicit predictors

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

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

so by constructing an interpolating polynomial $p_q(t)$ through a set of existing data, the initial guess at stage solutions may be approximated as

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

Denoting $[a, b]$ as the interval containing the data used to construct $p_q(t)$, and assuming forward integration from $a \to b$, it is typically the case that $t_{n,j} > b$. The dangers of using a polynomial interpolant to extrapolate values outside the interpolation interval are well-known, with higher-order polynomials and predictions further outside the interval resulting in the greatest potential inaccuracies.

Each prediction algorithm therefore constructs a different type of interpolant $p_q(t)$, as described below.

2.6.1 Trivial predictor

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

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

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

2.6.2 Maximum order predictor

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

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

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

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

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

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

- $q = 0$: constant interpolant

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

- $q = 1$: linear Lagrange interpolant

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

- $q = 2$: quadratic Hermite interpolant

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

- $q = 3$: cubic Hermite interpolant

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

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

2.6.3 Variable order predictor

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

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

2.6.4 Cutoff order predictor

This predictor follows a similar idea as the previous algorithm, but monitors the actual stage times to determine the polynomial interpolant to use for prediction:

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

2.6.5 Bootstrap predictor

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

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

For stages in which $c_j = 0$ for all previous stages $j = 0, \ldots, i-1$, and for the first stage of any time step $(i = 0)$, this method reduces to using the trivial predictor $z_i^{(0)} = y_{n-1}$. For stages having multiple precdding nonzero c_j, we choose the stage having largest c_j value, to minimize the amount of extrapolation induced through the prediction.

2.7 Time step adaptivity

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

$$\begin{aligned} \|y_n - y(t_n)\| &= C h_n^{q+1} + \mathcal{O}(h_n^{q+2}), \\ \|\tilde{y}_n - y(t_n)\| &= D h_n^{p+1} + \mathcal{O}(h_n^{p+2}), \end{aligned} \tag{2.15}$$

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

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

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

$$T_n = \beta\left(y_n - \tilde{y}_n\right) = \beta h_n M^{-1} \sum_{i=0}^{s} \left(b_i - \tilde{b}_i\right) \left(f_E(t_{n-1} + c_i h_n, z_i) + f_I(t_{n-1} + c_i h_n, z_i)\right). \tag{2.16}$$

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

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

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

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

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

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

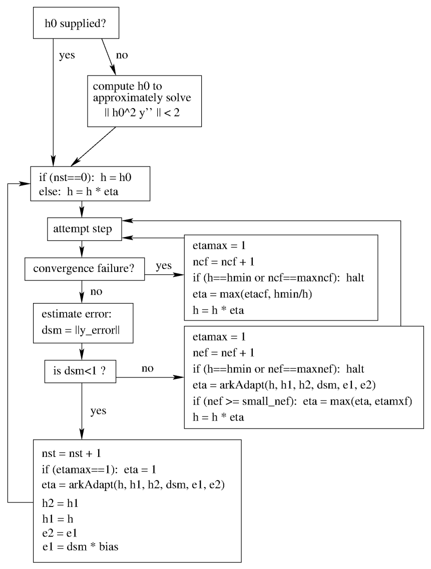
Here, the values of *etamx1* and *growth* may be modified by the user in the functions `ARKodeSetMaxFirstGrowth()` and `ARKodeSetMaxGrowth()`, respectively, or through the inputs *ADAPT_ETAMX1* and *ADAPT_GROWTH* to the function `FARKSETRIN()`.

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

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

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

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



2.7.1 Asymptotic error control

As mentioned above, ARKode adapts the step size in order to attain local errors within desired tolerances of the true solution. These adaptivity algorithms estimate the prospective step size h' based on the asymptotic local error estimates (2.15). We define the values ε_n, ε_{n-1} and ε_{n-2} as

$$\varepsilon_k \;\equiv\; \|T_k\| \;=\; \beta\|y_n - \tilde{y}_n\|,$$

corresponding to the local error estimates for three consecutive steps, $t_{n-3} \to t_{n-2} \to t_{n-1} \to t_n$. These local error history values are all initialized to 1.0 upon program initialization, to accomodate the few initial time steps of a calculation where some of these error estimates are undefined. With these estimates, ARKode implements a variety of error control algorithms, as specified in the subsections below.

PID controller

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

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

where the constants k_1, k_2 and k_3 default to 0.58, 0.21 and 0.1, respectively, though each may be changed via a call to the C/C++ function `ARKodeSetAdaptivityMethod()`, or to the Fortran function `FARKSETADAPTIVITYMETHOD()`. In this estimate, a floor of $\varepsilon > 10^{-10}$ is enforced to avoid division-by-zero errors.

PI controller

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

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

Here, the default values of k_1 and k_2 default to 0.8 and 0.31, respectively, though they may be changed via a call to `ARKodeSetAdaptivityMethod()` or `FARKSETADAPTIVITYMETHOD()`. As with the previous controller, at initialization $k_1 = k_2 = 1.0$ and the floor of 10^{-10} is enforced on the local error estimates.

I controller

The so-called I controller is the standard time adaptivity control algorithm in use by most available ODE solvers. It bases the prospective time step estimate entirely off of the current local error estimate,

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

By default, $k_1 = 1$, but that may be overridden by the user with the function `ARKodeSetAdaptivityMethod()` or the function `FARKSETADAPTIVITYMETHOD()`.

Explicit Gustafsson controller

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

$$h' \;=\; \begin{cases} h_1\; \varepsilon_1^{-1/p}, & \text{on the first step,} \\ h_n\; \varepsilon_n^{-k_1/p}\; \left(\varepsilon_n/\varepsilon_{n-1}\right)^{k_2/p}, & \text{on subsequent steps.} \end{cases} \tag{2.17}$$

The default values of k_1 and k_2 are 0.367 and 0.268, respectively, which may be changed bhy calling either `ARKodeSetAdaptivityMethod()` or `FARKSETADAPTIVITYMETHOD()`.

Implicit Gustafsson controller

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

$$h' = \begin{cases} h_1 \varepsilon_1^{-1/p}, & \text{on the first step,} \\ h_n \left(h_n/h_{n-1}\right) \varepsilon_n^{-k_1/p} \left(\varepsilon_n/\varepsilon_{n-1}\right)^{-k_2/p}, & \text{on subsequent steps.} \end{cases} \tag{2.18}$$

The algorithm parameters default to $k_1 = 0.98$ and $k_2 = 0.95$, but may be modified by the user with `ARKodeSetAdaptivityMethod()` or `FARKSETADAPTIVITYMETHOD()`.

ImEx Gustafsson controller

An ImEx version of these two preceding controllers is available in ARKode. This approach computes the estimates h_1' arising from equation (2.17) and the estimate h_2' arising from equation (2.18), and selects

$$h' = \frac{h}{|h|} \min\left\{|h_1'|, |h_2'|\right\}.$$

Here, equation (2.17) uses k_1 and k_2 with default values of 0.367 and 0.268, while equation (2.18) sets both parameters to the input k_3 that defaults to 0.95. All three of these parameters may be modified with the C/C++ function `ARKodeSetAdaptivityMethod()` or the Fortran function `FARKSETADAPTIVITYMETHOD()`.

User-supplied controller

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

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

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

2.8 Explicit stability

For problems that involve a nonzero explicit component, $f_E(t,y) \neq 0$, explicit and ImEx Runge-Kutta methods may benefit from addition user-supplied information regarding the explicit stability region. All ARKode adaptivity methods utilize estimates of the local error. It is often the case that such local error control will be sufficient for method stability, since unstable steps will typically exceed the error control tolerances. However, for problems in which $f_E(t,y)$ includes even moderately stiff components, and especially for higher-order integration methods, it may occur that a significant number of attempted steps will exceed the error tolerances. While these steps will automatically be recomputed, such trial-and-error may be costlier than desired. In these scenarios, a stability-based time step controller may also be useful.

Since the explicit stability region for any method depends on the problem under consideration, as it results from the eigenvalues of the linearized operator $\frac{\partial f_E}{\partial y}$, information on the maximum stable step size is not computed internally within ARKode. However, for many problems such information is readily available. For example, in an advection-diffusion calculation, f_I may contain the stiff diffusive components and f_E may contain the comparably nonstiff advection terms. In this scenario, an explicitly stable step h_{exp} would be predicted as one satisfying the Courant-Friedrichs-Lewy (CFL) stability condition,

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

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

In these scenarios, a user may supply a routine to predict this maximum explicitly stable step size, $|h_{\text{exp}}|$, by calling the C/C++ function `ARKodeSetStabilityFn()` or the Fortran function `FARKEXPSTABSET()`. If a value for $|h_{\text{exp}}|$ is supplied, it is compared against the value resulting from the local error controller, $|h_{\text{acc}}|$, and the step used by ARKode will satisfy

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

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

2.9 Mass matrix solver

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

$$Mx = b$$

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

$$M y_n = M y_{n-1} + h_n \sum_{i=0}^{s} b_i \left(f_E(t_{n,i}, z_i) + f_I(t_{n,i}, z_i) \right),$$

$$\Leftrightarrow$$

$$M(y_n - y_{n-1}) = h_n \sum_{i=0}^{s} b_i \left(f_E(t_{n,i}, z_i) + f_I(t_{n,i}, z_i) \right),$$

$$\Leftrightarrow$$

$$M\nu = h_n \sum_{i=0}^{s} b_i \left(f_E(t_{n,i}, z_i) + f_I(t_{n,i}, z_i) \right),$$

for the update $\nu = y_n - y_{n-1}$. Similarly, in computing the local temporal error estimate T_n from equation (2.16) we must solve systems of the form

$$M\,T_n = h \sum_{i=0}^{s} \left(b_i - \tilde{b}_i \right) \left(f_E(t_{n,i}, z_i) + f_I(t_{n,i}, z_i) \right).$$

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

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

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

At present, for DIRK and ARK problems using a dense or band solver for the Newton nonlinear iterations, the type of linear solver (dense or band) for the Newton systems $\mathcal{A}\delta = -G$ must match the type of linear solver used for these mass-matrix systems, since M is included inside $\mathcal{A}$. When direct methods (dense and band) are employed, the user must supply a routine to compute M in either dense or band form to match the structure of $\mathcal{A}$, using either the routine `ARKDlsDenseMassFn()` or `ARKDlsBandMassFn()`. When iterative methods are used, a routine must be supplied to perform the mass-matrix-vector product, Mv, through a call to the routine `ARKSpilsMassTimesVecFn()`. As with iterative solvers for the Newton systems, preconditioning may be applied to aid in solution of the mass matrix systems $Mx = b$.

We further note that non-identity mass matrices, $M \neq I$, are only supported by the C and C++ ARKode interfaces, although Fortran support is planned for the near future.

2.10 Rootfinding

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

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

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

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

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

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

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

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

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

CHAPTER

THREE

CODE ORGANIZATION

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

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing. The following is a list of the solver packages presently available:

- ARKode, a Runge-Kutta solver for stiff, nonstiff and multi-rate ODEs $M\dot{y} = f_E(t, y) + f_I(t, y)$;
- CVODE, a linear multistep solver for stiff and nonstiff ODEs $\dot{y} = f(t, y)$;
- CVODES, a linear multistep solver for stiff and nonstiff ODEs with sensitivity analysis capabilities;
- IDA, a linear multistep solver for differential-algebraic systems $F(t, y, \dot{y}) = 0$;
- IDAS, a linear multistep solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$.

3.1 ARKode organization

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

The overall organization of the ARKode package is as follows. The central integration module, implemented in the files `arkode.h`, `arkode_impl.h` and `arkode.c`, deals with the evaluation of integration stages, the nonlinear solver (if $f_I(t, y) \neq 0$), estimation of the local truncation error, selection of step size, and interpolation to user output points, among other issues. ARKode currently supports modified Newton, inexact Newton, and accelerated fixed-point solvers for these implicit problems. However, when using the Newton-based iterations, or when using a non-identity mass matrix $M \neq I$, ARKode has flexibility in the choice of method used to solve the linear sub-systems that arise. Therefore, for any user problem invoking the Newton solvers, or any user problem with $M \neq I$, one (or more) of the linear system solver modules should be specified by the user, which is then invoked as needed during the integration process.

For solving these linear systems, ARKode presently includes the following linear algebra modules, organized into two families. The *direct* family of linear solvers provides methods for the direct solution of linear systems with dense or banded matrices and includes:

- ARKDENSE: LU factorization and backsolving with dense matrices (using either an internal implementation or BLAS/LAPACK);
- ARKBAND: LU factorization and backsolving with banded matrices (using either an internal implementation or BLAS/LAPACK).

The *spils* family of linear solvers provides scaled preconditioned linear solvers and includes:

- ARKSPGMR: scaled preconditioned GMRES method;
- ARKSPBCG: scaled preconditioned Bi-CGStab method;
- ARKSPTFQMR: scaled preconditioned TFQMR method;
- ARKSPFGMR: scaled preconditioned flexible GMRES method;
- ARKPCG: preconditioned conjugate gradient method;

The set of linear solver modules distributed with ARKode is intended to be expanded in the future as new algorithms are developed, and may additionally be expanded through user-supplied linear solver modules, further described in the section *Providing Alternate Linear Solver Modules*.

In the case of the direct methods (ARKDENSE and ARKBAND), ARKode includes an algorithm to approximate the Jacobian using difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. In the case of the Krylov iterative methods (ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG), ARKode includes an algorithm to approximate the product between the Jacobian matrix and a vector, also using difference quotients. Again, the user has the option of supplying a routine for this operation. For the Krylov methods, preconditioning must be supplied by the user, in two phases: *setup* (preprocessing of Jacobian data) and *solve*. While there is no default choice of preconditioner analagous to the difference-quotient approximation in the direct case, the references [BH1989] and [B1992], together with the example and demonstration programs included with ARKode and CVODE, offer considerable assistance in building simple preconditioners.

Each ARKode linear solver module consists of four routines, devoted to

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

The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration process, and only as required to achieve convergence. The call list within the central ARKode module to each of the four associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

These modules are also decomposed in another way. With the exception of the modules interfacing to LAPACK linear solvers, each of the modules ARKDENSE, ARKBAND, ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARK-SPFGMR and ARKPCG is a set of interface routines built on top of a generic solver module, named DENSE, BAND, SPGMR, SPBCG, SPTFQMR, SPFGMR and PCG, respectively. The interfaces deal with the use of these methods in the ARKode context, whereas the generic solvers are independent of the context where they are used. This separation allows for any generic solver to be replaced by an improved version, with no necessity to revise the ARKode package structure.

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 vector data structure, NVECTOR_SERIAL, 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. There is no global data in the ARKode package, and so in this respect it is reentrant. State information specific to the linear solver is saved in a separate data structure, a pointer to which resides in the ARKode memory structure.

CHAPTER

FOUR

USING ARKODE FOR C AND C++ APPLICATIONS

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

The example programs described in the companion document [R2013] may be helpful. Those codes may be used as templates for new codes and are included in the ARKode package `examples` subdirectory.

Users with applications written in Fortran should see the chapter *FARKODE, an Interface Module for FORTRAN Applications*, that describes the Fortran/C interface module, and may look to the Fortran example programs also described in the companion document [R2013]. These codes are also located in the ARKode package `examples` directory.

The user should be aware that not all linear solver and preconditioning modules are compatible with all NVECTOR implementations. For example, NVECTOR_PARALLEL is not compatible with the direct dense or direct band linear solvers since these linear solver modules need to form the complete system Jacobian on a single processor. Specifically, the following ARKode modules can only be used with NVECTOR_SERIAL: ARKDENSE, ARKBAND (using either the internal or the LAPACK implementation) and ARKBANDPRE. Also, the preconditioner module ARKBBDPRE can only be used with NVECTOR_PARALLEL.

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

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

4.1 Access to library and header files

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

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

- `libdir/libsundials_arkode.lib`,
- `libdir/libsundials_nvec*.lib` (one or two files),

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

- `incdir/include/arkode`
- `incdir/include/sundials`
- `incdir/include/nvector`

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

4.2 Data Types

The `sundials_types.h` file contains the definition of the variable type `realtype`, which is used by the SUNDIALS solvers for all floating-point data. The type "realtype" can be set to `float`, `double`, or `long double`, depending on how SUNDIALS was installed (with the default being `double`). The user can change the precision of the SUNDIALS solvers' floating-point arithmetic at the configuration stage (see the section *ARKode Installation Procedure*).

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

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

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

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

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent, except for any calls to precision-specific standard math library functions. Users can, however, use the types `double`, `float`, or `long double` in their code (assuming that this usage is consistent with the size of `realtype` values that are passed to and from SUNDIALS). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, so long as the SUNDIALS libraries have been compiled using the same precision (for details see the section *ARKode Installation Procedure*).

SUNDIALS also defines a type "booleantype", that can have values `TRUE` and `FALSE`, which is used for logic arguments within the library.

4.3 Header Files

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

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

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

The calling program must also include an NVECTOR implementation header file (see the section *Vector Data Structures* for details). For the two NVECTOR implementations that are included in the ARKode package, the corresponding header files are:

- `nvector_serial.h`, which defines the serial implementation NVECTOR_SERIAL;
- `nvector_parallel.h`, which defines the parallel (MPI) implementation, NVECTOR_PARALLEL.

Note that both these files in turn include the header file `sundials_nvector.h` which defines the abstract `N_Vector` data type.

If the user includes a non-trivial implicit component to their ODE system, then each time step will require a nonlinear solver for the resulting systems of equations. ARKode allows an accelerated fixed point iteration and Newton-based iterations for this solver; if a Newton method is used then a linear solver module header file may also be required. Similarly, if the ODE system

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

involves a non-identity mass matrix $M \neq I$, then each time step will require a linear solver for systems of the form $Mx = b$. The header files corresponding to the various linear solvers built into ARKode, and that can be used with either the Newton solver or for mass-matrix solves, are:

- `arkode_dense.h`, which is used with the dense direct linear solver;
- `arkode_band.h`, which is used with the band direct linear solver;
- `arkode_lapack.h`, which is used with LAPACK implementations of dense or band direct linear solvers;
- `arkode_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver SPGMR;
- `arkode_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver SPBCG;
- `arkode_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov solver SPTFQMR.
- `arkode_spfgmr.h`, which is used with the scaled, preconditioned Flexible GMRES Krylov linear solver SPFGMR;
- `arkode_pcg.h`, which is used with the preconditioned conjugate gradient linear solver PCG;

The header files for the dense and banded linear solvers (both internal and LAPACK) include the file `arkode_direct.h`, which defines common functions. This in turn includes a file (`sundials_direct.h`) which defines the matrix type for these direct linear solvers (`DlsMat`), as well as various functions and macros for acting on and accessing entries of such matrices.

The header files for the Krylov iterative solvers each include `arkode_spils.h` which defines common functions and which in turn includes a header file (`sundials_iterative.h`) which enumerates the preconditioning type and the choices for the Gram-Schmidt orthogonalization process (for the SPGMR and SPFGMR solvers).

Other headers may be needed, according to the choice of preconditioner, etc. For example, if preconditioning for an iterative linear solver were performed using a block-diagonal matrix, the header `sundials_dense.h` may need to be included for access to the underlying generic dense linear solver to be used for preconditioning.

4.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of an IVP. Some steps are independent of the NVECTOR implementation used. Where this is not the case, usage specifications are given for the two implementations provided with ARKode: steps marked [P] correspond to NVECTOR_PARALLEL, while steps marked [S] correspond to NVECTOR_SERIAL.

1. [P] Initialize MPI

   Call `MPI_Init` to initialize MPI if used by the user's program.

2. Set problem dimensions

   [S] Set `N`, the problem size N.

   [P] Set `Nlocal`, the local vector length (the sub-vector length for this process); `N`, the global vector length (the problem size N, equaling the sum of all the values of `Nlocal` on the active set of processes).

3. Set vector of initial values

   To set the vector `y0` of initial values, use the appropriate functions defined by the particular NVECTOR implementation. If a `realtype` array `ydata` containing the initial values of y already exists, then make the call:

   [S] `y0 = N_VMake_Serial(N, ydata);`

   [P] `y0 = N_VMake_Parallel(comm, Nlocal, N, ydata);`

   Otherwise, make the call:

   [S] `y0 = N_VNew_Serial(N);`

   [P] `y0 = N_VNew_Parallel(comm, Nlocal, N);`

   and load initial values into the array accessed by:

   [S] `NV_DATA_S(y0)`

   [P] `NV_DATA_P(y0)`

   Here `comm` is the MPI communicator containing the set of active processes to be used (may be the MPI default, `MPI_COMM_WORLD`).

4. Create ARKode object

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

5. Initialize ARKode solver

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

6. Specify integration tolerances

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

7. Set optional inputs

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

8. Attach linear solver module

   If an implicit solve is required and a Newton-based iteration is chosen for the solver, initialize the linear solver module with one of the following calls (for details see the section *Linear solver specification functions*):

   [S] `ier = ARKDense(...);`

   [S] `ier = ARKBand(...);`

   [S] `ier = ARKLapackDense(...);`

[S] `ier = ARKLapackBand(...);`

`ier = ARKSpgmr(...);`

`ier = ARKSpbcg(...);`

`ier = ARKSptfqmr(...);`

`ier = ARKSpfgmr(...);`

`ier = ARKPcg(...);`

9. Set linear solver optional inputs

   Call `ARK*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See the section *Optional input functions* for details.

10. Attach mass matrix linear solver module

    If a non-identity mass matrix solve is required, initialize the linear mass matrix solver module with one of the following calls (for details see the section *Linear solver specification functions*):

    [S] `ier = ARKMassDense(...);`

    [S] `ier = ARKMassBand(...);`

    [S] `ier = ARKMassLapackDense(...);`

    [S] `ier = ARKMassLapackBand(...);`

    `ier = ARKMassSpgmr(...);`

    `ier = ARKMassSpbcg(...);`

    `ier = ARKMassSptfqmr(...);`

    `ier = ARKMassSpfgmr(...);`

    `ier = ARKMassPcg(...);`

11. Set mass matrix linear solver optional inputs

    Call `ARK*Set*` functions from the selected mass matrix linear solver module to change optional inputs specific to that linear solver. See the section *Optional input functions* for details.

12. Specify rootfinding problem

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

13. Advance solution in time

    For each point at which output is desired, call

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

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

14. Get optional outputs

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

15. Free solver memory

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

16. Deallocate memory for solution vector

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

    [S] `N_VDestroy_Serial(y);`

    [P] `N_VDestroy_Parallel(y);`

17. [P] Finalize MPI

    Call `MPI_Finalize` to terminate MPI.

4.5 User-callable functions

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

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

4.5.1 ARKode initialization and deallocation functions

void* **`ARKodeCreate`**()

This function creates an internal memory block for a problem to be solved by ARKode.

Arguments: None

Return value: If successful, a pointer to initialized problem memory of type `void*`, to be passed to `ARKodeInit()`. If unsuccessful, a `NULL` pointer will be returned, and an error message will be printed to `stderr`.

int **`ARKodeInit`**(void* *arkode_mem*, ARKRhsFn *fe*, ARKRhsFn *fi*, realtype *t0*, realtype *y0*)

This function allocates and initializes memory for a problem to to be solved by ARKode.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block (that was returned by `ARKodeCreate()`)
- *fe* – the name of the C function (of type `ARKRhsFn()`) defining the explicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$
- *fi* – the name of the C function (of type `ARKRhsFn()`) defining the implicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$
- *t0* – the initial value of t
- *y0* – the initial condition vector $y(t_0)$

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKode memory was `NULL`
- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

void **ARKodeFree** (void* *arkode_mem*)

This function frees the problem memory *arkode_mem* created by `ARKodeCreate()` and allocated by `ARKodeInit()`.

Arguments:

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

Return value: None

4.5.2 ARKode tolerance specification functions

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

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

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

whereas in the case of `ARKodeSVtolerances()` the vector components are given by

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

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

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

where N is the problem dimension.

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

int **ARKodeSStolerances** (void* *arkode_mem*, realtype *reltol*, realtype *abstol*)

This function specifies scalar relative and absolute tolerances.

Arguments:

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

Return value:

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

int **ARKodeSVtolerances** (void* *arkode_mem*, realtype *reltol*, N_Vector *abstol*)

This function specifies a scalar relative tolerance and a vector absolute tolerance (a potentially different absolute tolerance for each vector component).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *reltol* – scalar relative tolerance
- *abstol* – vector containing the absolute tolerances for each solution component

Return value:

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

int **`ARKodeWFtolerances`**(void* *arkode_mem*, ARKEwtFn *efun*)

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

Arguments:

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

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKode memory was `NULL`
- *ARK_NO_MALLOC* if the ARKode memory was not allocated by `ARKodeInit()`

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

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

whereas in the case of `ARKodeResVtolerance()` the vector components are given by

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

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

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

where N is the problem dimension.

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

int **`ARKodeResStolerance`**(void* *arkode_mem*, realtype *abstol*)

This function specifies a scalar absolute residual tolerance.

Arguments:

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

Return value:

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

int **`ARKodeResVtolerance`**(void* *arkode_mem*, N_Vector *rabstol*)

This function specifies a vector of absolute residual tolerances.

Arguments:

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

Return value:

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

int **`ARKodeResFtolerance`**(void* *arkode_mem*, ARKRwtFn *rfun*)

This function specifies a user-supplied function *rfun* to compute the residual weight vector `rwt`.

Arguments:

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

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKode memory was `NULL`
- *ARK_NO_MALLOC* if the ARKode memory was not allocated by `ARKodeInit()`

General advice on the choice of tolerances

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

1. The scalar relative tolerance `reltol` is to be set to control relative errors. So a value of 10^{-4} means that errors are controlled to .01%. We do not recommend using `reltol` larger than 10^{-3}. On the other hand, `reltol` should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 10^{-15} for double-precision).
2. The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector y may be so small that pure relative error control is meaningless. For example, if y_i starts at some nonzero value, but in time decays to zero, then pure relative error control on y_i makes no sense (and is overly costly) after y_i is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol`

should be a vector. For example, see the example problem `ark_robertson.c`, and the discussion of it in the ARKode Examples Documentation [R2013]. In that problem, the three components vary betwen 0 and 1, and have different noise levels; hence the `atols` vector therein. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

3. The residual absolute tolerances `rabstol` (whether scalar or vector) follow a similar explanation as for `abstol`, except that these should be set to the noise level of the equation components, i.e. the noise level of My. For problems in which $M = I$, it is recommended that `rabstol` be left unset, which will default to the already-supplied `abstol` values.
4. Finally, it is important to pick all the tolerance values conservately, because they control the error committed on each individual step. The final (global) errors are an accumulation of those per-step errors, where that accumulation factor is problem-dependent. A general rule of thumb is to reduce the tolerances by a factor of 10 from the actual desired limits on errors. I.e. if you want .01% relative accuracy (globally), a good choice for `reltol` is 10^{-5}. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

Advice on controlling unphysical negative values

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

1. The best way to control the size of unwanted negative computed values is with tighter absolute tolerances. Again this requires some knowledge of the noise level of these components, which may or may not be different for different components. Some experimentation may be needed.
2. If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in y returned by ARKode, with magnitude comparable to `abstol` or less, is equivalent to zero as far as the computation is concerned.
3. The user's right-hand side routines f_E and f_I should never change a negative value in the solution vector y to a non-negative value in attempt to "fix" this problem, since this can lead to numerical instability. If the f_E or f_I routines cannot tolerate a zero or negative value (e.g. because there is a square root or log), then the offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input y vector) for the purposes of computing $f_E(t, y)$ or $f_I(t, y)$.
4. Positivity and non-negativity constraints on components can be enforced by use of the recoverable error return feature in the user-supplied right-hand side functions, f_E and f_I. When a recoverable error is encountered, ARKode will retry the step with a smaller step size, which typically alleviates the problem. However, because this option involves some additional overhead cost, it should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.

4.5.3 Linear solver specification functions

As previously explained, the modified Newton iteration used in solving implicit systems within ARKode requires the solution of linear systems of the form

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

where

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

There are seven ARKode linear solvers currently available for this task: ARKDENSE, ARKBAND, ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG.

The first two linear solvers are direct solvers based on Gaussian elimination, and derive their names from the type of storage used for the approximate Jacobian J; ARKDENSE and ARKBAND work with dense and banded approximations to J, respectively. The SUNDIALS suite includes both internal implementations of these two linear solvers and interfaces to LAPACK implementations. Together, these linear solvers are referred to as *ARKDLS* (which stands for ARKode Direct Linear Solvers).

The last five ARKode linear solvers, ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG, are Krylov iterative solvers, which use scaled preconditioned GMRES, scaled preconditioned Bi-CGStab, scaled preconditioned TFQMR, scaled preconditioned flexible GMRES, and preconditioned conjugate gradient, respectively. Together, they are referred to as *ARKSPILS* (which stands for ARKode Scaled Preconditioned Iterative Linear Solvers).

With any of the Krylov methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all (except for ARKPCG that applies a single preconditioner in a symmetric manner). For the specification of a preconditioner, see the iterative linear solver portions of the sections *Optional input functions* and *User-supplied functions*.

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

To specify a ARKode linear solver, after the call to `ARKodeCreate()` but before any calls to `ARKode()`, the user's program must call one of the functions `ARKDense()`/`ARKLapackDense()`, `ARKBand()`/`ARKLapackBand()`, `ARKSpgmr()`, `ARKSpbcg()`, `ARKSptfqmr()`, `ARKSpfgmr()` or `ARKPcg()` as documented below. The first argument passed to these functions is the ARKode memory pointer returned by `ARKodeCreate()`. A call to one of the above solver specification functions links the main ARKode integrator to a linear solver and allows the user to specify parameters which are specific to that solver, such as the half-bandwidths in the `ARKBand()` case. The use of each of the linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the linear solver, as specified below.

In each case except the LAPACK direct solvers, the linear solver module used by ARKode is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted DENSE, BAND, SPGMR, SPBCG, SPTFQMR, SPFGMR and PCG, are described separately in the section *Linear Solvers in ARKode*.

int **ARKDense**(void* *arkode_mem*, long int *N*)

This function links the main ARKode integrator with the ARKDENSE linear solver. It's use requires inclusion of the header file `arkode_dense.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKDENSE linear solver is not compatible with all implementations of the NVECTOR module. Of the two nvector modules provided with SUNDIALS, only NVECTOR_SERIAL is compatible.

int **`ARKLapackDense`** (void* *arkode_mem*, int *N*)
This function links the main ARKode integrator with the ARKLAPACK linear solver module. It's use requires inclusion of the header file `arkode_lapack.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: Here *N* is restricted to be of type `int`, because of the corresponding type restriction in the LAPACK solvers.

int **`ARKBand`** (void* *arkode_mem*, long int *N*, long int *mupper*, long int *mlower*)
This function links the main ARKode integrator with the ARKBAND linear solver. It's use requires inclusion of the header file `arkode_band.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system
- *mupper* – the upper bandwidth of the band Jacobian approximation
- *mlower* – is the lower bandwidth of the band Jacobian approximation.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKBAND linear solver is not compatible with all implementations of the NVECTOR module. Of the two NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL is compatible.

The half-bandwidths are to be set such that the nonzero locations (i, j) in the banded (approximate) Jacobian satisfy *-mlower* $\leq j - i \leq$ *mupper*.

int **`ARKLapackBand`** (void* *arkode_mem*, int *N*, int *mupper*, int *mlower*)
This function links the main ARKode integrator with the ARKLAPACK linear solver using banded Jacobians. It's use requires inclusion of the header file `arkode_lapack.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system
- *mupper* – the upper bandwidth of the band Jacobian approximation

- *mlower* – is the lower bandwidth of the band Jacobian approximation.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: Here, each of *N*, *mupper* and *mlower* are restricted to be of type `int`, because of the corresponding type restriction in the LAPACK solvers.

int **`ARKSpgmr`** (void* *arkode_mem*, int *pretype*, int *maxl*)

This function links the main ARKode integrator with the ARKSPGMR linear solver. It's use requires inclusion of the header file `arkode_spgmr.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_spgmr.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPGMR solver. Pass 0 to use the default value of 5.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear systems.

int **`ARKSpbcg`** (void* *arkode_mem*, int *pretype*, int *maxl*)

This function links the main ARKode integrator with the ARKSPBCG linear solver. It's use requires inclusion of the header file `arkode_spbcgs.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_spbcgs.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPBCG solver. Pass 0 to use the default value of 5.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPBCG solver uses a scaled preconditioned Bi-CGStab iterative method to solve the linear systems.

int **`ARKSptfqmr`**(void* *arkode_mem*, int *pretype*, int *maxl*)

This function links the main ARKode integrator with the ARKSPTFQMR linear solver. It's use requires inclusion of the header file `arkode_sptfqmr.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_sptfqmr.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPTFMR solver. Pass 0 to use the default value of 5.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPTFQMR solver uses a scaled preconditioned TFQMR iterative method to solve the linear systems.

int **`ARKSpfgmr`**(void* *arkode_mem*, int *pretype*, int *maxl*)

This function links the main ARKode integrator with the ARKSPFGMR linear solver. It's use requires inclusion of the header file `arkode_spfgmr.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_spfgmr.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPFGMR solver. Pass 0 to use the default value of 5.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPFGMR solver uses a scaled preconditioned flexible GMRES iterative method to solve the linear systems.

int **ARKPcg** (void* *arkode_mem*, int *pretype*, int *maxl*)

This function links the main ARKode integrator with the ARKPCG linear solver. It's use requires inclusion of the header file `arkode_pcg.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – flag denoting whether to use preconditioning. If set to any of the enumeration constants *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH*, defined in `sundials_iterative.h` (already included by `arkode_pcg.h`), preconditioning will be enabled. Due to the symmetric form of PCG, there is no choice between left and right preconditioning.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKPCG solver. Pass 0 to use the default value of 5.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKPCG solver uses a preconditioned conjugate gradient iterative method to solve the linear systems.

4.5.4 Mass matrix solver specification functions

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

$$Mx = b.$$

The same solvers listed above in the section *Linear solver specification functions* may be used for this purpose: DENSE, BAND, SPGMR, SPBCG, SPTFQMR, SPFGMR and PCG. With any of the iterative solvers (SPGMR, SPBCG, SPTFQMR, SPFGMR and PCG), preconditioning can be applied. For the specification of a preconditioner, see the iterative linear solver portions of the sections *Optional input functions* and *User-supplied functions*. If preconditioning is to be performed, user-supplied functions should be used to define left and right preconditioner matrices P_1 and P_2 (either of which could be the identity matrix), such that the product $P_1 P_2$ approximates the mass matrix M.

To specify a mass matrix solver, after the call to `ARKodeCreate()` but before any calls to `ARKode()`, the user's program must call one of the functions `ARKMassDense()`/`ARKMassLapackDense()`, `ARKMassBand()`/`ARKMassLapackBand()`, `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassSpfgmr()` or `ARKMassPcg()` as documented below. The first argument passed to these functions is the ARKode memory pointer returned by `ARKodeCreate()`. A call to one of these solver specification functions links the mass matrix solve with the specified solver module, and allows the user to specify parameters which are specific to the desired solver. The use of each of the linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the linear solver, as specified below.

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

int **ARKMassDense** (void* *arkode_mem*, long int *N*, ARKDlsDenseMassFn *dmass*)
This function links the mass matrix solve with the ARKDENSE linear solver module, and specifies the dense mass matrix function. It's use requires inclusion of the header file `arkode_dense.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system.
- *dmass* – name of user-supplied dense mass matrix function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKDENSE linear solver is not compatible with all implementations of the NVECTOR module. Of the two nvector modules provided with SUNDIALS, only NVECTOR_SERIAL is compatible.

int **ARKMassLapackDense** (void* *arkode_mem*, int *N*, ARKDlsDenseMassFn *dmass*)
This function links the mass matrix solve with the ARKLAPACK linear solver module. It's use requires inclusion of the header file `arkode_lapack.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system.
- *dmass* – name of user-supplied dense mass matrix function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: Here *N* is restricted to be of type `int`, because of the corresponding type restriction in the LAPACK solvers.

int **ARKMassBand** (void* *arkode_mem*, long int *N*, long int *mupper*, long int *mlower*, ARKDlsBandMassFn *bmass*)
This function links the mass matrix solve with the ARKBAND linear solver module. It's use requires inclusion of the header file `arkode_band.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system.
- *mupper* – the upper bandwidth of the band mass matrix.
- *mlower* – is the lower bandwidth of the band mass matrix.
- *bmass* – name of user-supplied band mass matrix function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKBAND linear solver may not be compatible with the particular implementation of the NVECTOR module. Of the two NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL is compatible. The half-bandwidths are to be set such that the nonzero locations (i, j) in the banded mass matrix satisfy *-mlower* $\leq j - i \leq$ *mupper*.

At present, it is required that the band mass matrix have identical band structure to the Jacobian matrix. While this is typical of finite-element problems, if this is not true for a specific problem it can be handled by manually zero-padding the mass matrix.

int **`ARKMassLapackBand`**(void* *arkode_mem*, int *N*, int *mupper*, int *mlower*, ARKDlsBandMassFn *bmass*)

This function links the mass matrix solve with the ARKLAPACK linear solver module. It's use requires inclusion of the header file `arkode_lapack.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – the number of components in the ODE system.
- *mupper* – the upper bandwidth of the band mass matrix.
- *mlower* – is the lower bandwidth of the band mass matrix.
- *bmass* – name of user-supplied band mass matrix function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MEM_FAIL* if there was a memory allocation failure
- *ARKDLS_ILL_INPUT* if a required vector operation is missing

Notes: Here, each of *N*, *mupper* and *mlower* are restricted to be of type `int`, because of the corresponding type restriction in the LAPACK solvers.

At present, it is required that the band mass matrix have identical band structure to the Jacobian matrix. While this is typical of finite-element problems, if this is not true for a specific problem it can be handled by manually zero-padding the mass matrix.

int **`ARKMassSpgmr`**(void* *arkode_mem*, int *pretype*, int *maxl*, ARKSpilsMassTimesVecFn *mtimes*)

This function links the mass matrix solve with the ARKSPGMR linear solver module. It's use requires inclusion of the header file `arkode_spgmr.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_spgmr.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPGMR solver. Pass 0 to use the default value of 5.

- *mtimes* – user-defined mass-matrix-vector product function.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear systems.

int **`ARKMassSpbcg`**(void* *arkode_mem*, int *pretype*, int *maxl*, ARKSpilsMassTimesVecFn *mtimes*)

This function links the mass matrix solve with the ARKSPBCG linear solver module. It's use requires inclusion of the header file `arkode_spbcgs.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_spbcgs.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPBCG solver. Pass 0 to use the default value of 5.
- *mtimes* – user-defined mass-matrix-vector product function.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPBCG solver uses a scaled preconditioned Bi-CGStab iterative method to solve the linear systems.

int **`ARKMassSptfqmr`**(void* *arkode_mem*, int *pretype*, int *maxl*, ARKSpilsMassTimesVecFn *mtimes*)

This function links the mass matrix solve with the ARKSPTFQMR linear solver. It's use requires inclusion of the header file `arkode_sptfqmr.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_sptfqmr.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPTFMR solver. Pass 0 to use the default value of 5.
- *mtimes* – user-defined mass-matrix-vector product function.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPTFQMR solver uses a scaled preconditioned TFQMR iterative method to solve the linear systems.

int **`ARKMassSpfgmr`** (void* *arkode_mem*, int *pretype*, int *maxl*, ARKSpilsMassTimesVecFn *mtimes*)

This function links the mass matrix solve with the ARKSPFGMR linear solver. It's use requires inclusion of the header file `arkode_spfgmr.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of user preconditioning to be done. This must be one of the four enumeration constants *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH* defined in `sundials_iterative.h` (already included by `arkode_spfgmr.h`). These correspond to no preconditioning, left preconditioning only, right preconditioning only, and both left and right preconditioning, respectively.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKSPFGMR solver. Pass 0 to use the default value of 5.
- *mtimes* – user-defined mass-matrix-vector product function.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKSPFGMR solver uses a scaled preconditioned flexible GMRES iterative method to solve the linear systems.

int **`ARKMassPcg`** (void* *arkode_mem*, int *pretype*, int *maxl*, ARKSpilsMassTimesVecFn *mtimes*)

This function links the mass matrix solve with the ARKPCG linear solver. It's use requires inclusion of the header file `arkode_pcg.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – flag denoting whether to use preconditioning. If set to any of the enumeration constants *PREC_LEFT*, *PREC_RIGHT*, or *PREC_BOTH*, defined in `sundials_iterative.h` (already included by `arkode_pcg.h`), preconditioning will be enabled. Due to the symmetric form of PCG, there is no choice between left and right preconditioning.
- *maxl* – the maximum Krylov dimension. This is an optional input to the ARKPCG solver. Pass 0 to use the default value of 5.
- *mtimes* – user-defined mass-matrix-vector product function.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure
- *ARKSPILS_ILL_INPUT* if a required vector operation is missing

Notes: The ARKPCG solver uses a preconditioned conjugate gradient iterative method to solve the linear systems.

4.5.5 Rootfinding initialization function

As described in the section *Rootfinding*, while solving the IVP ARKode has the capability to find the roots of a set of user-defined functions. To activate the root-finding algorithm, call the following function:

int **ARKodeRootInit** (void* *arkode_mem*, int *nrtfn*, ARKRootFn *g*)
Initializes a rootfinding problem to be solved during the integration of the ODE system. It must be called after `ARKodeCreate()`, and before `ARKode()`.

Arguments:

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

Return value:

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

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

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

4.5.6 ARKode solver function

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

int **ARKode** (void* *arkode_mem*, realtype *tout*, N_Vector *yout*, realtype **tret*, int *itask*)
Integrates the ODE over an interval in t.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *tout* – the next time at which a computed solution is desired
- *yout* – the computed solution vector
- *tret* – the time reached by the solver (output)

- *itask* – a flag indicating the job of the solver for the next user step.

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

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

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

Return value:

- *ARK_SUCCESS* if successful
- *ARK_ROOT_RETURN* if ARKode succeeded, and found one or more roots. If *nrtfn* is greater than 1, call `ARKodeGetRootInfo()` to see which g_i were found to have a root at (**tret*).
- *ARK_TSTOP_RETURN* if ARKode succeeded and returned at *tstop*.
- *ARK_MEM_NULL* if the *arkode_mem* argument was `NULL`.
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if one of the inputs to ARKode is illegal, or some other input to the solver was either illegal or missing. Details will be provided in the error message. Typical causes of this failure:
  1. The tolerances have not been set.
  2. A component of the error weight vector became zero during internal time-stepping.
  3. The linear solver initialization function (called by the user after calling `ARKodeCreate()`) failed to set the linear solver-specific *lsolve* field in *arkode_mem*.
  4. A root of one of the root functions was found both at a point t and also very near t.
- *ARK_TOO_MUCH_WORK* if the solver took *mxstep* internal steps but could not reach *tout*. The default value for *mxstep* is *MXSTEP_DEFAULT = 500*.
- *ARK_TOO_MUCH_ACC* if the solver could not satisfy the accuracy demanded by the user for some internal step.
- *ARK_ERR_FAILURE* if error test failures occurred either too many times (*ark_maxnef*) during one internal time step or occurred with $|h| = h_{min}$.
- *ARK_CONV_FAILURE* if either convergence test failures occurred too many times (*ark_maxncf*) during one internal time step or occurred with $|h| = h_{min}$.
- *ARK_LINIT_FAIL* if the linear solver's initialization function failed.
- *ARK_LSETUP_FAIL* if the linear solver's setup routine failed in an unrecoverable manner.
- *ARK_LSOLVE_FAIL* if the linear solver's solve routine failed in an unrecoverable manner.
- *ARK_MASSINIT_FAIL* if the mass matrix solver's initialization function failed.
- *ARK_MASSSETUP_FAIL* if the mass matrix solver's setup routine failed.
- *ARK_MASSSOLVE_FAIL* if the mass matrix solver's solve routine failed.

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

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

All failure return values are negative and so testing the return argument for negative values will trap all ARKode failures.

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

4.5.7 Optional input functions

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

The optional inputs are grouped into the following categories:

- General solver options (*Optional inputs for ARKode*),
- IVP method solver options (*Optional inputs for IVP method selection*),
- Step adaptivity solver options (*Optional inputs for time step adaptivity*),
- Implicit stage solver options (*Optional inputs for implicit stage solves*),
- Dense linear solver options (*Direct linear solvers optional input functions*),
- Iterative linear solver options (*Iterative linear solvers optional input functions*).

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

We note that, on an error return, all of the optional input functions send an error message to the error handler function. We also note that all error return values are negative, so a test on the return arguments for negative values will catch all errors.

Optional inputs for ARKode

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

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

Arguments:

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

Return value:

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

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

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *dord* – requested polynomial order of accuracy

Return value:

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

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

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

Arguments:

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

Return value:

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

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

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

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

Arguments:

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

Return value:

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

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

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

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

int **`ARKodeSetErrHandlerFn`**(void* *arkode_mem*, ARKErrHandlerFn *ehfun*, void* *eh_data*)

Specifies the optional user-defined function to be used in handling error messages.

Arguments:

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

Return value:

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

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

int **`ARKodeSetInitStep`**(void* *arkode_mem*, realtype *hin*)

Specifies the initial time step size ARKode should use after initialization or reinitialization.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *hin* – value of the initial step to be attempted (≥ 0)

Return value:

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

Notes: Pass 0.0 to use the default value.

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

int **`ARKodeSetMaxHnilWarns`**(void* *arkode_mem*, int *mxhnil*)

Specifies the maximum number of messages issued by the solver to warn that $t + h = t$ on the next internal step, before ARKode will instead return with an error.

Arguments:

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

Return value:

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

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

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

int **`ARKodeSetMaxNumSteps`** (void* *arkode_mem*, long int *mxsteps*)

Specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time, before ARKode will return with an error.

Arguments:

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

Return value:

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

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

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

int **`ARKodeSetMaxErrTestFails`** (void* *arkode_mem*, int *maxnef*)

Specifies the maximum number of error test failures permitted in attempting one step, before ARKode will return with an error.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *maxnef* – maximum allowed number of error test failures (> 0)

Return value:

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

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

int **`ARKodeSetMaxStep`** (void* *arkode_mem*, realtype *hmax*)

Specifies the upper bound on the magnitude of the time step size.

Arguments:

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

Return value:

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

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

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

Arguments:

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

Return value:

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

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

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

Arguments:

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

Return value:

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

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

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

Arguments:

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

Return value:

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

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

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

Arguments:

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

Return value:

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

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

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

Optional inputs for IVP method selection

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

int **`ARKodeSetOrder`**(void* *arkode_mem*, int *ord*)

Specifies the order of accuracy for the integration method.

Arguments:

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

Return value:

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

Notes: For explicit methods, the allowed values are $2 \le ord \le 6$. For implicit methods, the allowed values are $2 \le ord \le 5$, and for IMEX methods the allowed values are $3 \le ord \le 5$. Any illegal input will result in the default value of 4.

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

int **`ARKodeSetImEx`**(void* *arkode_mem*)

Specifies that both the implicit and explicit portions of problem are enabled, and to use an additive Runge Kutta method.

Arguments:

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

Return value:

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

Notes: This is automatically deduced when neither of the function pointers *fe* or *fi* passed to `ARKodeInit()` are `NULL`, but may be set directly by the user if desired.

int **`ARKodeSetExplicit`**(void* *arkode_mem*)
Specifies that the implicit portion of problem is disabled, and to use an explicit RK method.

Arguments:

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

Return value:

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

Notes: This is automatically deduced when the function pointer *fi* passed to `ARKodeInit()` is `NULL`, but may be set directly by the user if desired.

int **`ARKodeSetImplicit`**(void* *arkode_mem*)
Specifies that the explicit portion of problem is disabled, and to use a diagonally implicit RK method.

Arguments:

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

Return value:

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

Notes: This is automatically deduced when the function pointer *fe* passed to `ARKodeInit()` is `NULL`, but may be set directly by the user if desired.

int **`ARKodeSetARKTables`**(void* *arkode_mem*, int *s*, int *q*, int *p*, realtype* *c*, realtype* *Ai*, realtype* *Ae*, realtype* *b*, realtype* *bembed*)
Specifies a customized Butcher table pair for the additive RK method.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *s* – number of stages in the RK method.
- *q* – global order of accuracy for the RK method.
- *p* – global order of accuracy for the embedded RK method.
- *c* – array (of length *s*) of stage times for the RK method.
- *Ai* – array of coefficients defining the implicit RK stages. This should be stored as a 1D array of size *s*s*, in row-major order.

- *Ae* – array of coefficients defining the explicit RK stages. This should be stored as a 1D array of size *s*s*, in row-major order.
- *b* – array of coefficients (of length *s*) defining the time step solution.
- *bembed* – array of coefficients (of length *s*) defining the embedded solution.

Return value:

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

Notes: This automatically calls `ARKodeSetImEx()`.

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

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

Both RK methods must share the same *c*, *b* and *bembed* coefficients.

The embedding *bembed* is required.

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *s* – number of stages in the RK method.
- *q* – global order of accuracy for the RK method.
- *p* – global order of accuracy for the embedded RK method.
- *c* – array (of length *s*) of stage times for the RK method.
- *A* – array of coefficients defining the RK stages. This should be stored as a 1D array of size *s*s*, in row-major order.
- *b* – array of coefficients (of length *s*) defining the time step solution.
- *bembed* – array of coefficients (of length *s*) defining the embedded solution.

Return value:

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

Notes: This automatically calls `ARKodeSetExplicit()`.

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

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

The embedding *bembed* is required.

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *s* – number of stages in the RK method.
- *q* – global order of accuracy for the RK method.
- *p* – global order of accuracy for the embedded RK method.
- *c* – array (of length *s*) of stage times for the RK method.
- *A* – array of coefficients defining the RK stages. This should be stored as a 1D array of size *s*s*, in row-major order.
- *b* – array of coefficients (of length *s*) defining the time step solution.
- *bembed* – array of coefficients (of length *s*) defining the embedded solution.

Return value:

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

Notes: This automatically calls `ARKodeSetImplicit()`.

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

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

The embedding *bembed* is required.

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *itable* – index of the DIRK Butcher table.
- *etable* – index of the ERK Butcher table.

Return value:

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

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

This automatically calls `ARKodeSetImEx()`.

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *etable* – index of the Butcher table.

Return value:

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

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

This automatically calls `ARKodeSetExplicit()`.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *itable* – index of the Butcher table.

Return value:

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

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

This automatically calls `ARKodeSetImplicit()`.

Optional inputs for time step adaptivity

The mathematical explanation of ARKode's time step adaptivity algorithm, including how each of the parameters below is used within the code, is provided in the section *Time step adaptivity*.

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *hfun* – name of user-supplied adaptivity function.
- *h_data* – pointer to user data passed to *hfun* every time it is called.

Return value:

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

Notes: This function should focus on accuracy-based time step estimation; for stability based time steps the function `ARKodeSetStabilityFn()` should be used instead.

int **`ARKodeSetAdaptivityMethod`**(void* *arkode_mem*, int *imethod*, int *idefault*, int *pq*, realtype* *adapt_params*)

Specifies the method (and associated parameters) used for time step adaptivity.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *imethod* – accuracy-based adaptivity method choice ($0 \leq imethod \leq 5$): 0 is PID, 1 is PI, 2 is I, 3 is explicit Gustafsson, 4 is implicit Gustafsson, and 5 is the ImEx Gustafsson.
- *idefault* – flag denoting whether to use default adaptivity parameters (1), or that they will be supplied in the *adapt_params* argument (0).
- *pq* – flag denoting whether to use the embedding order of accuracy *p* (0) or the method order of accuracy *q* (1) within the adaptivity algorithm. *p* is the ARKode default.
- *adapt_params[0]* – k_1 parameter within accuracy-based adaptivity algorithms.
- *adapt_params[1]* – k_2 parameter within accuracy-based adaptivity algorithms.
- *adapt_params[2]* – k_3 parameter within accuracy-based adaptivity algorithms.

Return value:

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

Notes: If custom parameters are supplied, they will be checked for validity against published stability intervals. If other parameter values are desired, it is recommended to instead provide a custom function through a call to `ARKodeSetAdaptivityFn()`.

int **`ARKodeSetCFLFraction`**(void* *arkode_mem*, realtype *cfl_frac*)

Specifies the fraction of the estimated explicitly stable step to use.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *cfl_frac* – maximum allowed fraction of explicitly stable step (default is 0.5).

Return value:

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

Notes: Any non-positive parameter will imply a reset to the default value.

int **`ARKodeSetErrorBias`**(void* *arkode_mem*, realtype *bias*)

Specifies the bias to be applied to the error estimates within accuracy-based adaptivity strategies.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *bias* – bias applied to error in accuracy-based time step estimation (default is 1.5).

Return value:

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

Notes: Any value below 1.0 will imply a reset to the default value.

int **`ARKodeSetFixedStepBounds`** (void* *arkode_mem*, realtype *lb*, realtype *ub*)
Specifies the step growth interval in which the step size will remain unchanged.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lb* – lower bound on window to leave step size fixed (default is 1.0).
- *ub* – upper bound on window to leave step size fixed (default is 1.5).

Return value:

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

Notes: Any interval *not* containing 1.0 will imply a reset to the default values.

int **`ARKodeSetMaxCFailGrowth`** (void* *arkode_mem*, realtype *etacf*)
Specifies the maximum step size growth factor upon a convergence failure on a stage solve within a step.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *etacf* – time step reduction factor on a nonlinear solver convergence failure (default is 0.25).

Return value:

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

Notes: Any value outside the interval $(0, 1]$ will imply a reset to the default value.

int **`ARKodeSetMaxEFailGrowth`** (void* *arkode_mem*, realtype *etamxf*)
Specifies the maximum step size growth factor upon multiple successive accuracy-based error failures in the solver.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *etamxf* – time step reduction factor on multiple error fails (default is 0.3).

Return value:

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

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value outside the interval $(0, 1]$ will imply a reset to the default value.

int **ARKodeSetMaxFirstGrowth** (void* *arkode_mem*, realtype *etamx1*)
Specifies the maximum allowed step size change following the very first integration step.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *etamx1* – maximum allowed growth factor after the first time step (default is 10000.0).

Return value:

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

Notes: Any value ≤ 1.0 will imply a reset to the default value.

int **ARKodeSetMaxGrowth** (void* *arkode_mem*, realtype *mx_growth*)
Specifies the maximum growth of the step size between consecutive steps in the integration process.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *growth* – maximum allowed growth factor between consecutive time steps (default is 20.0).

Return value:

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

Notes: Any value ≤ 1.0 will imply a reset to the default value.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *safety* – safety factor applied to accuracy-based time step (default is 0.96).

Return value:

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

Notes: Any non-positive parameter will imply a reset to the default value.

int **ARKodeSetSmallNumEFails** (void* *arkode_mem*, int *small_nef*)
Specifies the threshold for "multiple" successive error failures before the *etamxf* parameter from `ARKodeSetMaxEFailGrowth()` is applied.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *small_nef* – bound to determine 'multiple' for *etamxf* (default is 2).

Return value:

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

Notes: Any non-positive parameter will imply a reset to the default value.

int **`ARKodeSetStabilityFn`**(void* *arkode_mem*, ARKExpStabFn *EStab*, void* *estab_data*)

Sets the problem-dependent function to estimate a stable time step size for the explicit portion of the ODE system.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *EStab* – name of user-supplied stability function.
- *estab_data* – pointer to user data passed to *EStab* every time it is called.

Return value:

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

Notes: This function should return an estimate of the absolute value of the maximum stable time step for the explicit portion of the ODE system. It is not required, since accuracy-based adaptivity may be sufficient for retaining stability, but this can be quite useful for problems where the explicit right-hand side function $f_E(t, y)$ may contain stiff terms.

Optional inputs for implicit stage solves

The mathematical explanation for ARKode's nonlinear solver strategies, including how each of the parameters below is used within the code, is provided in the section *Nonlinear solver methods*.

Optional input	Function name	Default
Specify use of the fixed-point stage solver	`ARKodeSetFixedPoint()`	`FALSE`
Specify use of the Newton stage solver	`ARKodeSetNewton()`	`TRUE`
Specify linearly implicit f_I	`ARKodeSetLinear()`	`FALSE`
Specify nonlinearly implicit f_I	`ARKodeSetNonlinear()`	`TRUE`
Implicit predictor method	`ARKodeSetPredictorMethod()`	3
Maximum number of nonlinear iterations	`ARKodeSetMaxNonlinIters()`	3
Coefficient in the nonlinear convergence test	`ARKodeSetNonlinConvCoef()`	0.1
Nonlinear convergence rate constant	`ARKodeSetNonlinCRDown()`	0.3
Nonlinear residual divergence ratio	`ARKodeSetNonlinRDiv()`	2.3
Max change in step signaling new J	`ARKodeSetDeltaGammaMax()`	0.2
Max steps between calls to new J	`ARKodeSetMaxStepsBetweenLSet()`	20
Maximum number of convergence failures	`ARKodeSetMaxConvFails()`	10

int **`ARKodeSetFixedPoint`**(void* *arkode_mem*, long int *fp_m*)

Specifies that the implicit portion of the problem should be solved using the accelerated fixed-point solver instead of the modified Newton iteration, and provides the maximum dimension of the acceleration subspace.

Arguments:

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

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

Return value:

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

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

int **`ARKodeSetNewton`**(void* *arkode_mem*)

Specifies that the implicit portion of the problem should be solved using the modified Newton solver.

Arguments:

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

Return value:

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

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

int **`ARKodeSetLinear`**(void* *arkode_mem*)

Specifies that the implicit portion of the problem is linear.

Arguments:

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

Return value:

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

Notes: Tightens the linear solver tolerances and takes only a single Newton iteration. Only useful when used in combination with the modified Newton iteration (not the fixed-point solver).

int **`ARKodeSetNonlinear`**(void* *arkode_mem*)

Specifies that the implicit portion of the problem is nonlinear.

Arguments:

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

Return value:

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

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *method* – method choice ($0 \leq$ *method* ≤ 4):
  - 0 is the trivial predictor,
  - 1 is the maximum order (dense output) predictor,
  - 2 is the variable order predictor, that decreases the polynomial degree for more distant RK stages,
  - 3 is the cutoff order predictor, that uses the maximum order for early RK stages, and a first-order predictor for distant RK stages,
  - 4 is the bootstrap predictor, that uses a second-order predictor based on only information within the current step.

Return value:

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

Notes: The default value is 3. If *method* is set to an undefined value, the trivial predictor will be used.

int **`ARKodeSetMaxNonlinIters`**(void* *arkode_mem*, int *maxcor*)
Specifies the maximum number of nonlinear solver iterations permitted per RK stage within each time step.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *maxcor* – maximum allowed solver iterations per stage (> 0).

Return value:

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

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nlscoef* – coefficient in nonlinear solver convergence test (> 0.0).

Return value:

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

Notes: The default value is 0.1; set *nlscoef* ≤ 0 to specify this default.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *crdown* – nonlinear convergence rate estimation constant (default is 0.3).

Return value:

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

Notes: Any non-positive parameter will imply a reset to the default value.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *rdiv* – tolerance on nonlinear correction size ratio to declare divergence (default is 2.3).

Return value:

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

Notes: Any non-positive parameter will imply a reset to the default value.

int **ARKodeSetDeltaGammaMax** (void* *arkode_mem*, realtype *dgmax*)
Specifies a scaled step size ratio tolerance, beyond which the linear solver setup routine will be signaled.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *dgmax* – tolerance on step size ratio change before calling linear solver setup routine (default is 0.2).

Return value:

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

Notes: Any non-positive parameter will imply a reset to the default value.

int **ARKodeSetMaxStepsBetweenLSet** (void* *arkode_mem*, int *msbp*)
Specifies the maximum number of steps allowed between calls to the linear solver setup routine.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *msbp* – maximum number of time steps between linear solver setup calls (default is 20).

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **`ARKodeSetMaxConvFails`**(void* *arkode_mem*, int *maxncf*)

Specifies the maximum number of nonlinear solver convergence failures permitted during one step, before ARKode will return with an error.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *maxncf* – maximum allowed nonlinear solver convergence failures per step (> 0).

Return value:

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

Notes: The default value is 10; set *maxncf* ≤ 0 to specify this default.

Upon each convergence failure, ARKode will first call the Jacobian setup routine and try again (if a Newton method is used). If a convergence failure still occurs, the time step size is reduced by the factor *etacf* (set within `ARKodeSetMaxCFailGrowth()`).

Direct linear solvers optional input functions

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

Table: Optional inputs for ARKDLS

Optional input	Function name	Default
Dense Jacobian function	`ARKDlsSetDenseJacFn()`	`DQ`
Dense mass matrix function	`ARKDlsSetDenseMassFn()`	none
Band Jacobian function	`ARKDlsSetBandJacFn()`	`DQ`
Band mass matrix function	`ARKDlsSetBandMassFn()`	none

The ARKDENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(t, y)$. This function must be of type `ARKDlsDenseJacFn()`. The user can supply a custom dense Jacobian function, or use the default internal difference quotient approximation that comes with the ARKDENSE solver. To specify a user-supplied Jacobian function *djac*, ARKDENSE provides the function `ARKDlsSetDenseJacFn()`. The ARKDENSE solver passes the user data pointer to the dense Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The user data pointer may be specified through `ARKodeSetUserData()`.

Similarly, if the ODE system involves a non-identity mass matrix, $M \neq I$, the ARKDENSE solver needs a function to compute a dense approximation to the mass matrix $M(t)$. If the Newton linear systems are solved using ARKDENSE and the mass matrix systems are not, then the user must supply his/her own dense mass matrix function, *dmass*, since there is no default value. This function must be of type `ARKDlsDenseMassFn()`, and should be set using the function `ARKDlsSetDenseMassFn()`. We note that the ARKDENSE solver passes the user data pointer to the dense mass matrix function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied mass matrix function, without using global data in the program. The pointer user data may be specified through `ARKodeSetUserData()`.

int **ARKDlsSetDenseJacFn**(void* *arkode_mem*, ARKDlsDenseJacFn *djac*)

Specifies the dense Jacobian approximation routine to be used for a direct dense linear solver.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *djac* – name of user-supplied dense Jacobian approximation function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: By default, ARKDENSE uses an internal difference quotient function. If `NULL` is passed in for *djac*, this default is used.

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

int **ARKDlsSetDenseMassFn**(void* *arkode_mem*, ARKDlsDenseMassFn *dmass*)

Specifies the dense mass matrix approximation routine to be used for a direct dense linear solver.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *dmass* – name of user-supplied dense mass matrix approximation function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`

Notes: This routine must be called after the mass matrix solver has been initialized through a call to one of `ARKMassDense()`, `ARKMassLapackDense()`, `ARKMassBand()`, `ARKMassLapackBand()`, `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassSpfgmr()` or `ARKMassPcg()`.

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

Similarly, the ARKBAND solver needs a function to compute a banded approximation to the Jacobian matrix $J(t, y)$. This function must be of type `ARKDlsBandJacFn()`. The user can supply a custom banded Jacobian approximation function, or use the default internal difference quotient approximation that comes with the ARKBAND solver. To specify a user-supplied Jacobian function, *bjac*, ARKBAND provides the function `ARKDlsSetBandJacFn()`. The ARKBAND solver passes the user data pointer to the banded Jacobian approximation function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer user data may be specified through `ARKodeSetUserData()`.

Similarly, if the ODE system involves a non-identity mass matrix, $M \neq I$, the ARKBAND solver needs a function to compute a band approximation to the mass matrix $M(t)$. If the Newton linear systems are solved using ARKBAND and the mass matrix systems are not, then the user must supply his/her own band mass matrix function, *bmass*, since there is no default value. This function must be of type `ARKDlsBandMassFn()`, and should be set using the function `ARKDlsSetBandMassFn()`. We note that the ARKBAND solver passes the user data pointer to the band mass matrix function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied mass matrix function, without using global data in the program. The pointer user data may be specified through `ARKodeSetUserData()`.

int **ARKDlsSetBandJacFn**(void* *arkode_mem*, ARKDlsBandJacFn *bjac*)

Specifies the band Jacobian approximation routine to be used for a direct band linear solver.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *bjac* – name of user-supplied banded Jacobian approximation function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: By default, ARKBAND uses an internal difference quotient function. If `NULL` is passed in for *bjac*, this default is used.

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

int **ARKDlsSetBandMassFn**(void* *arkode_mem*, ARKDlsBandMassFn *bmass*)

Specifies the band mass matrix approximation routine to be used for a direct band linear solver.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *bmass* – name of user-supplied banded mass matrix approximation function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`

Notes: This routine must be called after the mass matrix solver has been initialized through a call to one of `ARKMassDense()`, `ARKMassLapackDense()`, `ARKMassBand()`, `ARKMassLapackBand()`, `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassSpfgmr()` or `ARKMassPcg()`.

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

Iterative linear solvers optional input functions

As described in the section *Linear solver methods*, when using one of the ARKSPILS iterative linear solvers, a user may supply a preconditioning operator to aid in solution of the system. This operator consists of two user-supplied functions, *psetup* and *psolve*, that are supplied to ARKode using either the function `ARKSpilsSetPreconditioner()` (for preconditioning the Newton system), or the function `ARKSpilsSetMassPreconditioner()` (for preconditioning the mass matrix system). The *psetup* function should handle evaluation and preprocessing of any Jacobian or mass-matrix data needed by the user's preconditioner solve function, *psolve*. The user data pointer received through `ARKodeSetUserData()` (or a pointer to `NULL` if user data was not specified) is passed to the *psetup* and *psolve* functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. If preconditioning is supplied for both the Newton and mass matrix linear systems, it is expected that the user will supply different *psetup* and *psolve* function for each.

Additionally, when solving the Newton linear systems, the ARKSPILS solvers require a *jtimes* function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. The user can supply a custom Jacobian-times-vector approximation function, or use the default internal difference quotient function that comes with the ARKSPILS solvers. A user-defined Jacobian-vector function must be of type `ARKSpilsJacTimesVecFn()` and can be specified through a call to `ARKSpilsSetJacTimesVecFn()` (see the section *User-supplied functions* for specification details). As with the preconditioner user-supplied 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 function *jtimes* each time it is called.

Similarly, if a problem involves a non-identity mass matrix, $M \neq I$, then the ARKSPILS solvers require a *mtimes* function to compute an approximation to the product between the mass matrix $M(t)$ and a vector v. This function must be user-supplied, since there is no default value. *mtimes* must be of type `ARKSpilsMassTimesVecFn()`. and can be specified through a call to `ARKSpilsSetMassTimesVecFn()`. If an ARKSPILS solver is also used for the mass matrix linear systems, then the *mtimes* function will already be provided in the call to `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassPcg()` or `ARKMassSpfgmr()`, so it does not need to be supplied a second time.

Table: Optional inputs for ARKSPILS

Optional input	**Function name**	**Default**
Jv function (*jtimes*)	`ARKSpilsSetJacTimesVecFn()`	`DQ`
Newton linear and nonlinear tolerance ratio	`ARKSpilsSetEpsLin()`	0.05
Newton Krylov subspace size *(a)*	`ARKSpilsSetMaxl()`	5
Newton Gram-Schmidt orthogonalization type *(b)*	`ARKSpilsSetGSType()`	classical GS
Newton preconditioning functions	`ARKSpilsSetPreconditioner()`	`NULL`, `NULL`
Newton preconditioning type	`ARKSpilsSetPrecType()`	none
Mv function (*mtimes*)	`ARKSpilsSetMassTimesVecFn()`	none
Mass matrix linear and nonlinear tolerance ratio	`ARKSpilsSetMassEpsLin()`	0.05
Mass matrix Krylov subspace size *(a)*	`ARKSpilsSetMassMaxl()`	5
Mass matrix Gram-Schmidt orthog. type *(b)*	`ARKSpilsSetMassGSType()`	classical GS
Mass matrix preconditioning functions	`ARKSpilsSetMassPreconditioner()`	`NULL`, `NULL`
Mass matrix preconditioning type	`ARKSpilsSetMassPrecType()`	none

(a) Only for ARKSPBCG, ARMSPTFQMR and ARKPCG

(b) Only for ARKSPGMR and ARKSPFGMR

int **`ARKSpilsSetJacTimesVecFn`**(void* *arkode_mem*, ARKSpilsJacTimesVecFn *jtimes*)

Specifies the Jacobian-times-vector function.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *jtimes* – user-defined Jacobian-vector product function.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: The default is to use an internal finite difference quotient. If `NULL` is passed to *jtimes*, this default function is used.

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

int **`ARKSpilsSetEpsLin`**(void* *arkode_mem*, realtype *eplifac*)

Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the linear iteration.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *eplifac* – linear convergence safety factor (≥ 0.0).

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

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

int **`ARKSpilsSetMaxl`**(void* *arkode_mem*, int *maxl*)

Resets the maximum Krylov subspace size, *maxl*, from the value previously set, when using the Bi-CGStab, TFQMR or PCG linear solver methods.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *maxl* – maximum dimension of the Krylov subspace.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: The maximum subspace dimension is initially set in the call to the linear solver specification function (see the section *Linear solver specification functions*). This function call is needed only if *maxl* is being changed from its previous value.

An input value *maxl* ≤ 0, gives the default value, 5.

This option is available only for the ARKSPBCG, ARKSPTFQMR and ARKPCG linear solvers.

int **`ARKSpilsSetGSType`**(void* *arkode_mem*, int *gstype*)

Specifies the type of Gram-Schmidt orthogonalization to be used with the ARKSPGMR or ARKSPFGMR linear solvers. This must be one of the two enumeration constants *MODIFIED_GS* or *CLASSICAL_GS* defined in `sundials_iterative.h` (already included by both `arkode_spgmr.h` and `arkode_spfgmr.h`). These correspond to using modified Gram-Schmidt and classical Gram-Schmidt, respectively.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *gstype* – type of Gram-Schmidt orthogonalization.

Return value:

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

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: The default value is *MODIFIED_GS*.

This option is available only for the ARKSPGMR and ARKSPFGMR linear solvers.

int **`ARKSpilsSetPreconditioner`**(void* *arkode_mem*, ARKSpilsPrecSetupFn *psetup*, ARKSpilsPrecSolveFn *psolve*)

Specifies the user-supplied preconditioner setup and solve functions.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *psetup* – user defined preconditioner setup function. Pass `NULL` if no setup is needed.
- *psolve* – user-defined preconditioner solve function.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

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

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

int **`ARKSpilsSetPrecType`**(void* *arkode_mem*, int *pretype*)

Resets the type of preconditioner, *pretype*, from the value previously set.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of preconditioning to use, must be one of *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT* or *PREC_BOTH*.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: The preconditioning type is initially set in the call to the linear solver's specification function (see the section *Linear solver specification functions*). This function call is needed only if *pretype* is being changed from its original value.

int **`ARKSpilsSetMassTimesVecFn`**(void* *arkode_mem*, ARKSpilsMassTimesVecFn *mtimes*)

Specifies the mass matrix-times-vector function.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *mtimes* – user-defined mass matrix-vector product function.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: This function must be called *after* the mass matrix solver has been initialized, through a call to one of `ARKMassDense()`, `ARKMassLapackDense()`, `ARKMassBand()` or `ARKMassLapackBand()`. It is only required if the mass matrix solver is not iterative, since *mtimes* will already be supplied to one of `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassSpfgmr()` or `ARKMassPcg()`.

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

int **`ARKSpilsSetMassEpsLin`**(void* *arkode_mem*, realtype *eplifac*)

Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the mass matrix linear iteration.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *eplifac* – linear convergence safety factor (≥ 0.0).

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: This must be called *after* the iterative mass matrix solver has been initialized, through a call to one of `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassSpfgmr()` or `ARKMassPcg()`.

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

int **`ARKSpilsSetMassMaxl`**(void* *arkode_mem*, int *maxl*)

Resets the maximum mass matrix Krylov subspace size, *maxl*, from the value previously set, when using the Bi-CGStab, TFQMR or PCG linear solver methods.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *maxl* – maximum dimension of the mass matrix Krylov subspace.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: This must be called *after* the iterative mass matrix solver has been initialized, through a call to one of `ARKMassSpbcg()`, `ARKMassSptfqmr()` or `ARKMassPcg()`.

The maximum subspace dimension is initially set in the call to the linear mass matrix solver specification function. This function call is needed only if *maxl* is being changed from its previous value.

An input value *maxl* ≤ 0, gives the default value, 5.

This option is available only for the ARKSPBCG, ARKSPTFQMR and ARKPCG linear solvers.

int **`ARKSpilsSetMassGSType`**(void* *arkode_mem*, int *gstype*)

Specifies the type of Gram-Schmidt orthogonalization to be used with the ARKSPGMR or ARKSPFGMR linear mass matrix solvers. This must be one of the two enumeration constants *MODIFIED_GS* or *CLASSICAL_GS* defined in `sundials_iterative.h` (already included by `arkode_spgmr.h` and `arkode_spfgmr.h`). These correspond to using modified Gram-Schmidt and classical Gram-Schmidt, respectively.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *gstype* – type of Gram-Schmidt orthogonalization.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: This must be called *after* the iterative mass matrix solver has been initialized, through a call to one of `ARKMassSpgmr()` or `ARKMassSpfgmr()`.

The default value is *MODIFIED_GS*.

This option is available only for the ARKSPGMR and ARKSPFGMR linear solvers.

int **`ARKSpilsSetMassPreconditioner`**(void* *arkode_mem*, ARKSpilsMassPrecSetupFn *psetup*, ARKSpilsMassPrecSolveFn *psolve*)

Specifies the mass matrix preconditioner setup and solve functions.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *psetup* – user defined preconditioner setup function. Pass `NULL` if no setup is to be done.
- *psolve* – user-defined preconditioner solve function.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: This function must be called *after* the iterative mass matrix solver has been initialized, through a call to one of `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassSpfgmr()` or `ARKMassPcg()`.

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

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

int **`ARKSpilsSetMassPrecType`**(void* *arkode_mem*, int *pretype*)

Resets the type of mass matrix preconditioner, *pretype*, from the value previously set.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *pretype* – the type of preconditioning to use, must be one of *PREC_NONE*, *PREC_LEFT*, *PREC_RIGHT* or *PREC_BOTH*.

Return value:

- *ARKSPILS_SUCCESS* if successful.
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.
- *ARKSPILS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.
- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: This function must be called *after* the iterative mass matrix solver has been initialized, through a call to one of `ARKMassSpgmr()`, `ARKMassSpbcg()`, `ARKMassSptfqmr()`, `ARKMassSpfgmr()` or `ARKMassPcg()`.

The preconditioning type is initially set in the call to the mass matrix solver's specification function. This function call is needed only if *pretype* is being changed from its original value.

Rootfinding optional input functions

The following functions can be called to set optional inputs to control the rootfinding algorithm, the mathematics of which are described in the section *Rootfinding*.

Optional input	Function name	Default
Direction of zero-crossings to monitor	`ARKodeSetRootDirection()`	both
Disable inactive root warnings	`ARKodeSetNoInactiveRootWarn()`	enabled

int **`ARKodeSetRootDirection`**(void* *arkode_mem*, int* *rootdir*)

Specifies the direction of zero-crossings to be located and returned.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *rootdir* – state array of length *nrtfn*, the number of root functions g_i, as specified in the call to the function `ARKodeRootInit()`. If `rootdir[i] == 0` then crossing in either direction for g_i should be reported. A value of +1 or -1 indicates that the solver should report only zero-crossings where g_i is increasing or decreasing, respectively.

Return value:

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

Notes: The default behavior is to monitor for both zero-crossing directions.

int **`ARKodeSetNoInactiveRootWarn`**(void* *arkode_mem*)

Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

Arguments:

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

Return value:

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

Notes: ARKode will not report the initial conditions as a possible zero-crossing (assuming that one or more components g_i are zero at the initial time). However, if it appears that some g_i is identically zero at the initial time (i.e., g_i is zero at the initial time *and* after the first step), ARKode will issue a warning which can be disabled with this optional input function.

4.5.8 Interpolated output function

An optional function `ARKodeGetDky()` is available to obtain additional output values. This function should only be called after a successful return from `ARKode()`, as it provides interpolated values either of y or of its derivatives (up to the 3rd derivative) interpolated to any value of t in the last internal step taken by `ARKode()`. Internally, this *dense output* algorithm is identical to the algorithm used for the maximum order implicit predictors, described in the section *Maximum order predictor*, except that derivatives of the polynomial model may be evaluated upon request.

int **ARKodeGetDky**(void* *arkode_mem*, realtype *t*, int *k*, N_Vector *dky*)

Computes the *k*-th derivative of the function y at the time *t*, i.e. $\frac{d^{(k)}}{dt^{(k)}}y(t)$, for values of the independent variable satisfying $t_n - h_n \le t \le t_n$, with t_n as current internal time reached, and h_n is the last internal step size successfully used by the solver. The user may request *k* in the range {0,1,2,3}. This routine uses an interpolating polynomial of degree *max(dord, k)*, where *dord* is the argument provided to `ARKodeSetDenseOrder()`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *t* – the value of the independent variable at which the derivative is to be evaluated.
- *k* – the derivative order requested.
- *dky* – output vector (must be allocated by the user).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_BAD_K* if *k* is not in the range {0,1,2,3}.
- *ARK_BAD_T* if *t* is not in the interval $[t_n - h_n, t_n]$
- *ARK_BAD_DKY* if the *dky* vector was `NULL`
- *ARK_MEM_NULL* if the ARKode memory is `NULL`

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

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

4.5.9 Optional output functions

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

1. General ARKode output routines are in the subsection *Main solver optional output functions*,
2. ARKode implicit solver output routines are in the subsection *Implicit solver optional output functions*,
3. Output routines regarding root-finding results are in the subsection *Rootfinding optional output functions*,
4. Dense linear solver output routines are in the subsection *Direct linear solvers optional output functions* and
5. Iterative linear solver output routines are in the subsection *Iterative linear solvers optional output functions*.

Following each table, we elaborate on each function.

Some of the optional outputs, especially the various counters, can be very useful in determining the efficiency of various methods inside the `ARKode()` solver. For example:

- The counters *nsteps*, *nfe_evals* and *nfi_evals* provide a rough measure of the overall cost of a given run, and can be compared between runs with different solver options to suggest which set of options is the most efficient.
- The ratio *nniters/nsteps* measures the performance of the nonlinear iteration in solving the nonlinear systems at each stage, providing a measure of the degree of nonlinearity in the problem. Typical values of this for a Newton solver on a general problem range from 1.1 to 1.8.
- When using a Newton nonlinear solver, the ratio *njevals/nniters* (in the case of a direct linear solver), and the ratio *npevals/nniters* (in the case of an iterative linear solver) can measure the overall degree of nonlinearity in the problem, since these are updated infrequently, unless the Newton method convergence slows.
- When using a Newton nonlinear solver, the ratio *njevals/nniters* (when using a direct linear solver), and the ratio *nliters/nniters* (when using an iterative linear solver) can indicate the quality of the approximate Jacobian or preconditioner being used. For example, if this ratio is larger for a user-supplied Jacobian or Jacobian-vector product routine than for the difference-quotient routine, it can indicate that the user-supplied Jacobian is inaccurate.
- The ratio *expsteps/accsteps* can measure the quality of the ImEx splitting used, since a higher-quality splitting will be dominated by accuracy-limited steps.
- The ratio *nsteps/step_attempts* can measure the quality of the time step adaptivity algorithm, since a poor algorithm will result in more failed steps, and hence a lower ratio.

It is therefore recommended that users retrieve and output these statistics following each run, and take some time to investigate alternate solver options that will be more optimal for their particular problem of interest.

Main solver optional output functions

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

int **`ARKodeGetWorkSpace`** (void* *arkode_mem*, long int* *lenrw*, long int* *leniw*)

Returns the ARKode real and integer workspace sizes.

Arguments:

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

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

Return value:

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

int **`ARKodeGetNumSteps`** (void* *arkode_mem*, long int* *nsteps*)
Returns the cumulative number of internal steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nsteps* – number of steps taken in the solver.

Return value:

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

int **`ARKodeGetNumExpSteps`** (void* *arkode_mem*, long int* *expsteps*)
Returns the cumulative number of stability-limited steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *expsteps* – number of stability-limited steps taken in the solver.

Return value:

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

int **`ARKodeGetNumAccSteps`** (void* *arkode_mem*, long int* *accsteps*)
Returns the cumulative number of accuracy-limited steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *accsteps* – number of accuracy-limited steps taken in the solver.

Return value:

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

int **`ARKodeGetNumStepAttempts`** (void* *arkode_mem*, long int* *step_attempts*)
Returns the cumulative number of steps attempted by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *step_attempts* – number of steps attempted by solver.

Return value:

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

int **ARKodeGetNumRhsEvals** (void* *arkode_mem*, long int* *nfe_evals*, long int* *nfi_evals*)
Returns the number of calls to the user's right-hand side functions, f_E and f_I (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nfe_evals* – number of calls to the user's $f_E(t, y)$ function.
- *nfi_evals* – number of calls to the user's $f_I(t, y)$ function.

Return value:

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

Notes: The *nfi_evals* value does not account for calls made to f_I by a linear solver or preconditioner module.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *netfails* – number of error test failures.

Return value:

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *hinused* – actual value of initial step size.

Return value:

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

Notes: Even if the value of the initial integration step was specified by the user through a call to `ARKodeSetInitStep()`, this value may have been changed by ARKode to ensure that the step size fell within the prescribed bounds ($h_{min} \le h_0 \le h_{max}$), or to satisfy the local error test condition, or to ensure convergence of the nonlinear solver.

int **ARKodeGetLastStep** (void* *arkode_mem*, realtype* *hlast*)
Returns the integration step size taken on the last successful internal step.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *hlast* – step size taken on the last internal step.

Return value:

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *hcur* – step size to be attempted on the next internal step.

Return value:

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *tcur* – current internal time reached.

Return value:

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

int **ARKodeGetCurrentButcherTables**(void* *arkode_mem*, int* *s*, int* *q*, int* *p*, realtype* *Ai*, realtype* *Ae*, realtype* *c*, realtype* *b*, realtype* *bembed*)
Returns the explicit and implicit Butcher tables currently in use by the solver.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *s* – number of stages in the method.
- *q* – global order of accuracy of the method.
- *p* – global order of accuracy of the embedding.
- *Ai* – coefficients of DIRK method.
- *Ae* – coefficients of ERK method.
- *c* – array of internal stage times.
- *b* – array of solution coefficients.
- *bembed* – array of embedding coefficients.

Return value:

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

Notes: The user must allocate space for *Ae* and *Ai* of size `ARK_S_MAX*ARK_S_MAX`, and for *c*, *b* and *bembed* of size `ARK_S_MAX` prior to calling this function.

int **ARKodeGetTolScaleFactor**(void* *arkode_mem*, realtype* *tolsfac*)
Returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

Arguments:

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

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

Return value:

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *eweight* – solution error weights at the current time.

Return value:

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

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *ele* – vector of estimated local truncation errors.

Return value:

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

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

The values returned in *ele* are valid only if `ARKode()` returned a non-negative value.

The *ele* vector, together with the *eweight* vector from `ARKodeGetErrWeights()`, can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the RMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nsteps* – number of steps taken in the solver.
- *expsteps* – number of stability-limited steps taken in the solver.
- *accsteps* – number of accuracy-limited steps taken in the solver.
- *step_attempts* – number of steps attempted by the solver.
- *nfe_evals* – number of calls to the user's $f_E(t, y)$ function.

- *nfi_evals* – number of calls to the user's $f_I(t, y)$ function.
- *nlinsetups* – number of linear solver setup calls made.
- *netfails* – number of error test failures.
- *hinused* – actual value of initial step size.
- *hlast* – step size taken on the last internal step.
- *hcur* – step size to be attempted on the next internal step.
- *tcur* – current internal time reached.

Return value:

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

char ***`ARKodeGetReturnFlagName`**(long int *flag*)

Returns the name of the ARKode constant corresponding to *flag*.

Arguments:

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

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

Implicit solver optional output functions

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

int **`ARKodeGetNumLinSolvSetups`**(void* *arkode_mem*, long int* *nlinsetups*)

Returns the number of calls made to the linear solver's setup routine (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nlinsetups* – number of linear solver setup calls made.

Return value:

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

int **`ARKodeGetNumMassSolves`**(void* *arkode_mem*, long int* *nMassSolves*)

Returns the number of calls made to the mass matrix solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nMassSolves* – number of mass matrix solves made.

Return value:

- *ARK_SUCCESS* if successful

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nniters* – number of nonlinear iterations performed.

Return value:

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

int **`ARKodeGetNumNonlinSolvConvFails`** (void* *arkode_mem*, long int* *nncfails*)
Returns the number of nonlinear solver convergence failures that have occurred (so far).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nncfails* – number of nonlinear convergence failures.

Return value:

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

int **`ARKodeGetNonlinSolvStats`** (void* *arkode_mem*, long int* *nniters*, long int* *nncfails*)
Returns all of the nonlinear solver statistics in a single call.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nniters* – number of nonlinear iterations performed.
- *nncfails* – number of nonlinear convergence failures.

Return value:

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

Rootfinding optional output functions

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

int **`ARKodeGetRootInfo`** (void* *arkode_mem*, int* *rootsfound*)
Returns an array showing which functions were found to have a root.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *rootsfound* – array of length *nrtfn* with the indices of the user functions g_i found to have a root. For $i = 0 \dots$ *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 ARKode memory was `NULL`

Notes: The user must allocate space for *rootsfound* prior to calling this function.

For the components of g_i for which a root was found, the sign of `rootsfound[i]` indicates the direction of zero-crossing. A value of +1 indicates that g_i is increasing, while a value of -1 indicates a decreasing g_i.

int **`ARKodeGetNumGEvals`**(void* *arkode_mem*, long int* *ngevals*)

Returns the cumulative number of calls made to the user's root function g.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *ngevals* – number of calls made to g so far.

Return value:

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

Direct linear solvers optional output functions

The following optional outputs are available from the ARKDLS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the mass matrix routine, number of calls to the implicit right-hand side routine for finite-difference Jacobian approximation, and last return value from an ARKDLS function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix LS (for Linear Solver) or MLS (for Mass Linear Solver) has been added here (e.g. *lenrwLS*).

Optional output	**Function name**
Size of real and integer workspaces	`ARKDlsGetWorkSpace()`
Size of mass real and integer workspaces	`ARKDlsGetMassWorkSpace()`
No. of Jacobian evaluations	`ARKDlsGetNumJacEvals()`
No. of mass matrix evaluations	`ARKDlsGetNumMassEvals()`
No. of *fi* calls for finite diff. Jacobian evals	`ARKDlsGetNumRhsEvals()`
Last return flag from a linear solver function	`ARKDlsGetLastFlag()`
Last return flag from a mass matrix solver function	`ARKDlsGetLastMassFlag()`
Name of constant associated with a return flag	`ARKDlsGetReturnFlagName()`

int **`ARKDlsGetWorkSpace`**(void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)

Returns the real and integer workspace used by the ARKDLS linear solver (ARKDENSE or ARKBAND).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lenrwLS* – the number of `realtype` values in the ARKDLS workspace.
- *leniwLS* – the number of integer values in the ARKDLS workspace.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: For the ARKDENSE linear solver, in terms of the problem size n, the actual size of the real workspace is $2n^2$ `realtype` words, and the actual size of the integer workspace is n integer words. For the ARKBAND linear solver, in terms of n and the Jacobian lower and upper half-bandwidths m_L and m_U, the actual size of the real workspace is $(2m_U + 3m_L + 2)n$ `realtype` words, and the actual size of the integer workspace is n integer words.

int **ARKDlsGetMassWorkSpace**(void* *arkode_mem*, long int* *lenrwMLS*, long int* *leniwMLS*)

Returns the real and integer workspace used by the ARKDLS mass matrix linear solver (ARKDENSE or ARKBAND).

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lenrwMLS* – the number of `realtype` values in the ARKDLS workspace.
- *leniwMLS* – the number of integer values in the ARKDLS workspace.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: For the ARKDENSE linear solver, in terms of the problem size n, the actual size of the real workspace is $2n^2$ `realtype` words, and the actual size of the integer workspace is n integer words. For the ARKBAND linear solver, in terms of n and the Jacobian lower and upper half-bandwidths m_L and m_U, the actual size of the real workspace is $(2m_U + 3m_L + 2)n$ `realtype` words, and the actual size of the integer workspace is n integer words.

int **ARKDlsGetNumJacEvals**(void* *arkode_mem*, long int* *njevals*)

Returns the number of calls made to the ARKDLS (dense or band) Jacobian approximation routine.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *njevals* – number of calls to the Jacobian function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

int **ARKDlsGetNumMassEvals**(void* *arkode_mem*, long int* *nmevals*)

Returns the number of calls made to the ARKDLS (dense or band) mass matrix construction routine.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nmevals* – number of calls to the mass matrix function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nfevalsLS* – the number of calls made to the user-supplied f_I function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The value of *nfevalsLS* is incremented only if the default internal difference quotient function is used.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lsflag* – the value of the last return flag from an ARKDLS function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: If the ARKDENSE setup function failed (i.e. `ARKode()` returned *ARK_LSETUP_FAIL*), then the value of *lsflag* is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix. For all other failures, *lsflag* is negative.

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *mlsflag* – the value of the last return flag from an ARKDLS mass matrix solver function.

Return value:

- *ARKDLS_SUCCESS* if successful
- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKDLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: If the ARKDENSE setup function failed (i.e. `ARKode()` returned *ARK_LSETUP_FAIL*), then the value of *lsflag* is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) mass matrix. For all other failures, *lsflag* is negative.

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

Arguments:

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

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

Iterative linear solvers optional output functions

The following optional outputs are available from the ARKSPILS modules: workspace requirements, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector product routine, number of calls to the mass-matrix-vector product routine, number of calls to the implicit right-hand side routine for finite-difference Jacobian-vector product approximation, and last return value from a linear solver function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix LS (for Linear Solver) or MLS (for Mass Linear Solver) has been added here (e.g. *lenrwLS*).

Optional output	**Function name**
Size of real and integer workspaces	`ARKSpilsGetWorkSpace()`
No. of preconditioner evaluations	`ARKSpilsGetNumPrecEvals()`
No. of preconditioner solves	`ARKSpilsGetNumPrecSolves()`
No. of linear iterations	`ARKSpilsGetNumLinIters()`
No. of linear convergence failures	`ARKSpilsGetNumConvFails()`
No. of Jacobian-vector product evaluations	`ARKSpilsGetNumJtimesEvals()`
No. of *fi* calls for finite diff. Jacobian-vector evals.	`ARKSpilsGetNumRhsEvals()`
Last return from a linear solver function	`ARKSpilsGetLastFlag()`
Size of real and integer mass matrix solver workspaces	`ARKSpilsGetMassWorkSpace()`
No. of mass matrix preconditioner evaluations	`ARKSpilsGetNumMassPrecEvals()`
No. of mass matrix preconditioner solves	`ARKSpilsGetNumMassPrecSolves()`
No. of mass matrix linear iterations	`ARKSpilsGetNumMassIters()`
No. of mass matrix solver convergence failures	`ARKSpilsGetNumMassConvFails()`
No. of mass-matrix-vector product evaluations	`ARKSpilsGetNumMtimesEvals()`
Last return from a mass matrix solver function	`ARKSpilsGetLastMassFlag()`
Name of constant associated with a return flag	`ARKSpilsGetReturnFlagName()`

int **`ARKSpilsGetWorkSpace`**(void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)

Returns the global sizes of the ARKSPILS real and integer workspaces.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lenrwLS* – the number of `realtype` values in the ARKSPILS workspace.
- *leniwLS* – the number of integer values in the ARKSPILS workspace.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: In terms of the problem size n and maximum Krylov subspace size m, the actual size of the real workspace is roughly: $(m+5)n + m(m+4) + 1$ `realtype` words for ARKSPGMR, $9n$ `realtype` words for ARKSPBCG, $11n$ `realtype` words for ARKSPTFQMR, $(2m+4)n + m(m+4) + 1$ `realtype` words for ARKSPFGMR, and $4n + 1$ `realtype` words for ARKPCG.

In a parallel setting, the above values are global, summed over all processors.

int **`ARKSpilsGetNumPrecEvals`**(void* *arkode_mem*, long int* *npevals*)

Returns the total number of preconditioner evaluations, i.e. the number of calls made to *psetup* with *jok* = `FALSE`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *npevals* – the current number of calls to *psetup*.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumPrecSolves`**(void* *arkode_mem*, long int* *npsolves*)

Returns the number of calls made to the preconditioner solve function, *psolve*.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *npsolves* – the number of calls to *psolve*.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumLinIters`**(void* *arkode_mem*, long int* *nliters*)

Returns the cumulative number of linear iterations.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nliters* – the current number of linear iterations.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumConvFails`**(void* *arkode_mem*, long int* *nlcfails*)

Returns the cumulative number of linear convergence failures.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nlcfails* – the current number of linear convergence failures.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumJtimesEvals`**(void* *arkode_mem*, long int* *njvevals*)

Returns the cumulative number of calls made to the Jacobian-vector function, *jtimes*.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *njvevals* – the current number of calls to *jtimes*.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumRhsEvals`**(void* *arkode_mem*, long int* *nfevalsLS*)

Returns the number of calls to the user-supplied implicit right-hand side function f_I for finite difference Jacobian-vector product approximation.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nfevalsLS* – the number of calls to the user implicit right-hand side function.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The value *nfevalsLS* is incremented only if the default internal difference quotient function is used.

int **`ARKSpilsGetLastFlag`**(void* *arkode_mem*, long int* *lsflag*)

Returns the last return value from an ARKSPILS routine.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lsflag* – the value of the last return flag from an ARKSPILS function.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: If the ARKSPILS setup function failed (`ARKode()` returned *ARK_LSETUP_FAIL*), then *lsflag* will be *SPGMR_PSET_FAIL_UNREC*, *SPBCG_PSET_FAIL_UNREC*, *SPTFQMR_PSET_FAIL_UNREC*, *SPFGMR_PSET_FAIL_UNREC*, or *PCG_PSET_FAIL_UNREC*.

If the ARKSPGMR solve function failed (`ARKode()` returned *ARK_LSOLVE_FAIL*), then *lsflag* contains the error return flag from SpgmrSolve and will be one of: *SPGMR_MEM_NULL*, indicating that the SPGMR memory is `NULL`; *SPGMR_ATIMES_FAIL_UNREC*, indicating an unrecoverable failure in the $J * v$ function; *SPGMR_PSOLVE_FAIL_UNREC*, indicating that the preconditioner solve function *psolve* failed unrecoverably; *SPGMR_GS_FAIL*, indicating a failure in the Gram-Schmidt procedure; or *SPGMR_QRSOL_FAIL*, indicating that the matrix R was found to be singular during the QR solve phase.

If the ARKSPBCG solve function failed (`ARKode()` returned *ARK_LSOLVE_FAIL*), then *lsflag* contains the error return flag from SpbcgSolve and will be one of: *SPBCG_MEM_NULL*, indicating that the SPBCG memory is `NULL`; *SPBCG_ATIMES_FAIL_UNREC*, indicating an unrecoverable failure in the $J * v$ function; or *SPBCG_PSOLVE_FAIL_UNREC*, indicating that the preconditioner solve function *psolve* failed unrecoverably.

If the ARKSPTFQMR solve function failed (`ARKode()` returned *ARK_LSOLVE_FAIL*), then *lsflag* contains the error return flag from SptfqmrSolve and will be one of: *SPTFQMR_MEM_NULL*, indicating that the SPTFQMR memory is `NULL`; *SPTFQMR_ATIMES_FAIL_UNREC*, indicating an unrecoverable failure in the $J * v$ function; or *SPTFQMR_PSOLVE_FAIL_UNREC*, indicating that the preconditioner solve function *psolve* failed unrecoverably.

If the ARKSPFGMR solve function failed (`ARKode()` returned *ARK_LSOLVE_FAIL*), then *lsflag* contains the error return flag from SpfgmrSolve and will be one of: *SPFGMR_MEM_NULL*, indicating that the SPFGMR memory is `NULL`; *SPFGMR_ATIMES_FAIL_UNREC*, indicating an unrecoverable failure in the $J * v$ function; *SPFGMR_PSOLVE_FAIL_UNREC*, indicating that the preconditioner solve function *psolve* failed unrecoverably; *SPFGMR_GS_FAIL*, indicating a failure in the Gram-Schmidt procedure; or *SPFGMR_QRSOL_FAIL*, indicating that the matrix R was found to be singular during the QR solve phase.

If the ARKPCG solve function failed (`ARKode()` returned *ARK_LSOLVE_FAIL*), then *lsflag* contains the error return flag from PcgSolve and will be one of: *PCG_MEM_NULL*, indicating that the PCG memory is `NULL`; *PCG_ATIMES_FAIL_UNREC*, indicating an unrecoverable failure in the $J * v$ function; or *PCG_PSOLVE_FAIL_UNREC*, indicating that the preconditioner solve function *psolve* failed unrecoverably.

char ***ARKSpilsGetReturnFlagName**(long int *lsflag*)

Returns the name of the ARKSPILS constant corresponding to *lsflag*.

Arguments:

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

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

int **ARKSpilsGetMassWorkSpace**(void* *arkode_mem*, long int* *lenrwMLS*, long int* *leniwMLS*)

Returns the global sizes of the ARKSPILS real and integer workspaces.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lenrwMLS* – the number of `realtype` values in the ARKSPILS workspace.
- *leniwMLS* – the number of integer values in the ARKSPILS workspace.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: In terms of the problem size n and maximum Krylov subspace size m, the actual size of the real workspace is roughly: $(m + 5)n + m(m + 4) + 1$ `realtype` words for ARKSPGMR, $9n$ `realtype` words for ARKSPBCG, $11n$ `realtype` words for ARKSPTFQMR, $(2m + 4)n + m(m + 4) + 1$ `realtype` words for ARKSPFGMR, and $4n + 1$ `realtype` words for ARKPCG.

In a parallel setting, the above values are global, summed over all processors.

int **ARKSpilsGetNumMassPrecEvals**(void* *arkode_mem*, long int* *nmpevals*)

Returns the total number of mass matrix preconditioner evaluations, i.e. the number of calls made to *psetup*.

Arguments:

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

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

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumMassPrecSolves`**(void* *arkode_mem*, long int* *nmpsolves*)

Returns the number of calls made to the mass matrix preconditioner solve function, *psolve*.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nmpsolves* – the number of calls to *psolve*.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumMassIters`**(void* *arkode_mem*, long int* *nmiters*)

Returns the cumulative number of mass matrix solver iterations.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nmiters* – the current number of mass matrix solver linear iterations.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumMassConvFails`**(void* *arkode_mem*, long int* *nmcfails*)

Returns the cumulative number of mass matrix solver convergence failures.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nmcfails* – the current number of mass matrix solver convergence failures.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **`ARKSpilsGetNumMtimesEvals`**(void* *arkode_mem*, long int* *nmvevals*)

Returns the cumulative number of calls made to the mass-matrix-vector product function, *mtimes*.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nmvevals* – the current number of calls to *mtimes*.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *msflag* – the value of the last return flag from an ARKSPILS mass matrix solver function.

Return value:

- *ARKSPILS_SUCCESS* if successful
- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The values of *msflag* for each of the various solvers will match those described above for the function `ARKSpilsGetLastFlag()`.

4.5.10 ARKode reinitialization function

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

The use of `ARKodeReInit()` requires that the number of Runge Kutta stages, denoted by *s*, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the method order q and the problem type (explicit, implicit, ImEx) are left unchanged. If there are changes to the linear solver specifications, the user should make the appropriate ARK*Set* calls, as described in the section *Linear solver specification functions*.

int **`ARKodeReInit`**(void* *arkode_mem*, ARKRhsFn *fe*, ARKRhsFn *fi*, realtype *t0*, N_Vector *y0*)
Provides required problem specifications and reinitializes ARKode.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *fe* – the name of the C function (of type `ARKRhsFn()`) defining the explicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$.
- *fi* – the name of the C function (of type `ARKRhsFn()`) defining the implicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$.
- *t0* – the initial value of t.
- *y0* – the initial condition vector $y(t_0)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKode memory was `NULL`
- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

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

4.5.11 ARKode system resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatially-adaptive PDE simulations under a method-of-lines approach), the ARKode integrator may be "resized" between integration steps, through calls to the `ARKodeResize()` function. This function modifies ARKode's internal memory structures to use the new problem size, without destruction of the temporal adaptivity heuristics. It is assumed that the dynamical time scales before and after the vector resize will be comparable, so that all time-stepping heuristics prior to calling `ARKodeResize()` remain valid after the call. If instead the dynamics should be recomputed from scratch, the ARKode memory structure should be deleted with a call to `ARKodeFree()`, and recreated with calls to `ARKodeCreate()` and `ARKodeInit()`.

To aid in the vector resize operation, the user can supply a vector resize function that will take as input a vector with the previous size, and transform it in-place to return a corresponding vector of the new size. If this function (of type `ARKVecResizeFn()`) is not supplied (i.e. is set to `NULL`), then all existing vectors internal to ARKode will be destroyed and re-cloned from the new input vector.

In the case that the dynamical time scale should be modified slightly from the previous time scale, an input *hscale* is allowed, that will rescale the upcoming time step by the specified factor. If a value *hscale* ≤ 0 is specified, the default of 1.0 will be used.

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

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

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *ynew* – the newly-sized solution vector, holding the current dependent variable values $y(t_0)$.
- *hscale* – the desired scaling factor for the dynamical time scale (i.e. the next step will be of size *h*hscale*).
- *t0* – the current value of the independent variable t_0 (this must be consistent with *ynew*.
- *resize* – the user-supplied vector resize function (of type `ARKVecResizeFn()`.
- *resize_data* – the user-supplied data structure to be passed to *resize* when modifying internal ARKode vectors.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKode memory was `NULL`
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: If an error occurred, `ARKodeResize()` also sends an error message to the error handler function.

Resizing the linear solver

When using any of the built-in 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()`. For each of the built-in ARKDLS and ARKSPILS linear solvers, the specification call itself (e.g. `ARKDense()` or `ARKSpgmr()`) will internally destroy the solver-specific memory prior to re-allocation.

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()`.

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.

Note: For an example of `ARKodeResize()` usage, see the supplied serial C example problem, `ark_heat1D_adapt.c`.

4.6 User-supplied functions

The user-supplied functions for ARKode consist of:

- at least one function defining the ODE (required),
- a function that handles error and warning messages (optional),
- a function that provides the error weight vector (optional),
- a function that provides the residual weight vector (optional),
- a function that handles adaptive time step error control (optional),
- a function that handles explicit time step stability (optional),
- a function that defines the root-finding problem(s) to solve (optional),
- a function that provides Jacobian-related information for the linear solver, if a Newton-based nonlinear iteration is chosen (optional),
- one or two functions that define the preconditioner for use in any of the Krylov iterative algorithms, if a Newton-based nonlinear iteration and iterative linear solver are chosen (optional), and
- if the problem involves a non-identity mass matrix $M \neq I$:
  - a function that provides mass-matrix-related information for the linear and mass matrix solvers (required),
  - one or two functions that define the mass matrix preconditioner for use in an iterative mass matrix solver is chosen (optional), and
- a function that handles vector resizing operations, if the underlying vector structure supports resizing (as opposed to deletion/recreation), and if the user plans to call `ARKodeResize()` (optional).

4.6.1 ODE right-hand side

The user must supply at least one function of type `ARKRhsFn()` to specify the explicit and/or implicit portions of the ODE system:

typedef int **`(*ARKRhsFn)`** (realtype *t*, N_Vector *y*, N_Vector *ydot*, void* *user_data*)

These functions compute the ODE right-hand side for a given value of the independent variable t and state vector y.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of teh dependent variable vector, $y(t)$.
- *ydot* – the output vector that forms a portion of the ODE RHS $f_E(t,y) + f_I(t,y)$.
- *user_data* – the *user_data* pointer that was passed to `ARKodeSetUserData()`.

Return value: An *ARKRhsFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKode will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *ARK_RHSFUNC_FAIL* is returned).

Notes: Allocation of memory for *ydot* is handled within ARKode. A recoverable failure error return from the *ARKRhsFn* is typically used to flag a value of the dependent variable y that is "illegal" in some way (e.g., negative where only a nonnegative value is physically meaningful). If such a return is made, ARKode will attempt to recover (possibly repeating the nonlinear iteration, or reducing the step size) in order to avoid this recoverable error return. There are some situations in which recovery is not possible even if the right-hand side function returns a recoverable error flag. One is when this occurs at the very first call to the *ARKRhsFn* (in which case ARKode returns *ARK_FIRST_RHSFUNC_ERR*). Another is when a recoverable error is reported by *ARKRhsFn* after the integrator completes a successful stage, in which case ARKode returns *ARK_UNREC_RHSFUNC_ERR*).

4.6.2 Error message handler function

As an alternative to the default behavior of directing error and warning messages to the file pointed to by *errfp* (see `ARKodeSetErrFile()`), the user may provide a function of type `ARKErrHandlerFn()` to process any such messages.

typedef void **`(*ARKErrHandlerFn)`** (int *error_code*, const char* *module*, const char* *function*, char* *msg*, void* *user_data*)

This function processes error and warning messages from ARKode and is sub-modules.

Arguments:

- *error_code* – the error code.
- *module* – the name of the ARKode module reporting the error.
- *function* – the name of the function in which the error occurred.
- *msg* – the error message.
- *user_data* – a pointer to user data, the same as the *eh_data* parameter that was passed to `ARKodeSetErrHandlerFn()`.

Return value: An *ARKErrHandlerFn* function has no return value.

Notes: *error_code* is negative for errors and positive (*ARK_WARNING*) for warnings. If a function that returns a pointer to memory encounters an error, it sets *error_code* to 0.

4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `ARKEwtFn()` to compute a vector *ewt* containing the weights in the WRMS norm $\|v\|_{WRMS} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(ewt_i\, v_i\right)^2\right)^{1/2}$. These weights will be used in place of those defined in the section *Choice of norm*.

typedef int **`(*ARKEwtFn)`** (N_Vector *y*, N_Vector *ewt*, void* *user_data*)

This function computes the WRMS error weights for the vector y.

Arguments:

- *y* – the dependent variable vector at which the weight vector is to be computed.
- *ewt* – the output vector containing the error weights.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.

Return value: An *ARKEwtFn* function must return 0 if it successfully set the error weights, and -1 otherwise.

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

The error weight vector must have all components positive. It is the user's responsibility to perform this test and return -1 if it is not satisfied.

4.6.4 Residual weight function

As an alternative to providing the scalar or vector absolute residual tolerances (when the IVP units differ from the solution units), the user may provide a function of type `ARKRwtFn()` to compute a vector *rwt* containing the weights in the WRMS norm $\|v\|_{WRMS} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(rwt_i\; v_i\right)^2\right)^{1/2}$. These weights will be used in place of those defined in the section *Choice of norm*.

typedef int **(*ARKRwtFn)** (N_Vector *y*, N_Vector *rwt*, void* *user_data*)
This function computes the WRMS residual weights for the vector y.

Arguments:

- *y* – the dependent variable vector at which the weight vector is to be computed.
- *rwt* – the output vector containing the residual weights.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.

Return value: An *ARKRwtFn* function must return 0 if it successfully set the residual weights, and -1 otherwise.

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

The residual weight vector must have all components positive. It is the user's responsibility to perform this test and return -1 if it is not satisfied.

4.6.5 Time step adaptivity function

As an alternative to using one of the built-in time step adaptivity methods for controlling solution error, the user may provide a function of type `ARKAdaptFn()` to compute a target step size h for the next integration step. These steps should be chosen as the maximum value such that the error estimates remain below 1.

typedef int **(*ARKAdaptFn)** (N_Vector *y*, realtype *t*, realtype *h1*, realtype *h2*, realtype *h3*, realtype *e1*, realtype *e2*, realtype *e3*, int *q*, int *p*, realtype* *hnew*, void* *user_data*)
This function implements a time step adaptivity algorithm that chooses h satisfying the error tolerances.

Arguments:

- *y* – the current value of the dependent variable vector, $y(t)$.
- *t* – the current value of the independent variable.
- *h1* – the current step size, $t_m - t_{m-1}$.
- *h2* – the previous step size, $t_{m-1} - t_{m-2}$.

- *h3* – the step size $t_{m-2} - t_{m-3}$.
- *e1* – the error estimate from the current step, m.
- *e2* – the error estimate from the previous step, $m - 1$.
- *e3* – the error estimate from the step $m - 2$.
- *q* – the global order of accuracy for the integration method.
- *p* – the global order of accuracy for the embedding.
- *hnew* – the output value of the next step size.
- *user_data* – a pointer to user data, the same as the *h_data* parameter that was passed to `ARKodeSetAdaptivityFn()`.

Return value: An *ARKAdaptFn* function should return 0 if it successfuly set the next step size, and a non-zero value otherwise.

4.6.6 Explicit stability function

A user may supply a function to predict the maximum stable step size for the explicit portion of the ImEx system, $f_E(t, y)$. While the accuracy-based time step adaptivity algorithms may be sufficient for retaining a stable solution to the ODE system, these may be inefficient if $f_E(t, y)$ contains moderately stiff terms. In this scenario, a user may provide a function of type `ARKExpStabFn()` to provide this stability information to ARKode. This function must set the scalar step size satisfying the stability restriction for the upcoming time step. This value will subsequently be bounded by the user-supplied values for the minimum and maximum allowed time step, and the accuracy-based time step.

typedef int **(*ARKExpStabFn)**(N_Vector *y*, realtype *t*, realtype* *hstab*, void* *user_data*)

This function predicts the maximum stable step size for the explicit portions of the ImEx ODE system.

Arguments:

- *y* – the current value of the dependent variable vector, $y(t)$.
- *t* – the current value of the independent variable
- *hstab* – the output value with the absolute value of the maximum stable step size.
- *user_data* – a pointer to user data, the same as the *estab_data* parameter that was passed to `ARKodeSetStabilityFn()`.

Return value: An *ARKExpStabFn* function should return 0 if it successfully set the upcoming stable step size, and a non-zero value otherwise.

Notes: If this function is not supplied, or if it returns *hstab* ≤ 0.0, then ARKode will assume that there is no explicit stability restriction on the time step size.

4.6.7 Rootfinding function

If a rootfinding problem is to be solved during the integration of the ODE system, the user must supply a function of type `ARKRootFn()`.

typedef int **(*ARKRootFn)**(realtype *t*, N_Vector *y*, realtype* *gout*, void* *user_data*)

This function implements a vector-valued function $g(t, y)$ such that the roots of the *nrtfn* components $g_i(t, y)$ are sought.

Arguments:

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

- *y* – the current value of the dependent variable vector, $y(t)$.
- *gout* – the output array, of length *nrtfn*, with components $g_i(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.

Return value: An *ARKRootFn* function should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and ARKode returns *ARK_RTFUNC_FAIL*).

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

4.6.8 Jacobian information (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is used (i.e., `ARKDense()` or `ARKLapackDense()` is called in Step 8 of the section *A skeleton of the user's main program*), the user may provide a function of type `ARKDlsDenseJacFn()` to provide the Jacobian approximation.

typedef int **(*ARKDlsDenseJacFn)**(long int *N*, realtype *t*, N_Vector *y*, N_Vector *fy*, DlsMat *Jac*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the dense Jacobian $J = \frac{\partial f_I}{\partial y}$ (or an approximation to it).

Arguments:

- *N* – the size of the ODE system.
- *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 dense Jacobian matrix (of type `DlsMat`).
- *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 ARKDlsDenseJacFn as temporary storage or work space.

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

Notes: 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). Only nonzero elements need to be loaded into *Jac* because *Jac* is set to the zero matrix before the call to the Jacobian function. The type of *Jac* is `DlsMat`.

The accessor macros `DENSE_ELEM` and `DENSE_COL` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `DlsMat` type. `DENSE_ELEM(J,i,j)` references the `(i,j)`-th element of the dense matrix `J` (for `i`, `j` between 0 and N-1). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to *N*, the Jacobian element $J_{m,n}$ can be set using the statement `DENSE_ELEM(J, m-1, n-1)` $= J_{m,n}$. Alternatively, `DENSE_COL(J,j)` returns a pointer to the first element of the `j`-th column of `J` (for `j` ranging from 0 to *N*-1), and the elements of the `j`-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements `col_n = DENSE_COL(J, n-1); col_n[m-1]` $= J_{m,n}$. For large problems, it is more efficient to use `DENSE_COL` than to use `DENSE_ELEM`. Note that both of these macros number rows and columns starting from 0.

The `DlsMat` type and accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in the section *Linear Solvers in ARKode*.

If the user's *ARKDenseJacFn* function uses difference quotient approximations, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, use the ARKodeGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

For the sake of uniformity, the argument *N* is of type `long int`, even in the case that the LAPACK dense solver is to be used.

4.6.9 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (i.e. `ARKBand()` or `ARKLapackBand()` is called in Step 8 of the section *A skeleton of the user's main program*), the user may provide a function of type `ARKDlsBandJacFn()` to provide the Jacobian approximation.

typedef int **(*ARKDlsBandJacFn)**(long int *N*, long int *mupper*, long int *mlower*, realtype *t*, N_Vector *y*, N_Vector *fy*, DlsMat *Jac*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the banded Jacobian $J = \frac{\partial f_I}{\partial y}$ (or an approximation to it).

Arguments:

- *N* – the size of the ODE system.
- *mlower*, *mupper* – the lower and upper half-bandwidths of the Jacobian.
- *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 dense Jacobian matrix (of type `DlsMat`).
- *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 ARKDlsBandJacFn as temporary storage or work space.

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

Notes: A user-supplied banded Jacobian function must load the band matrix *Jac* of type `DlsMat` with the elements of the Jacobian $J(t, y)$ at the point (t, y). Only nonzero elements need to be loaded into *Jac* because *Jac* is initialized to the zero matrix before the call to the Jacobian function.

The accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` allow the user to read and write band matrix elements without making specific references to the underlying representation of the `DlsMat` type. `BAND_ELEM(J, i, j)` references the `(i,j)`-th element of the band matrix `J`, counting from 0. This macro is meant for use in small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to *N* with (m, n) within the band defined by *mupper* and *mlower*, the Jacobian element $J_{m,n}$ can be loaded using the statement `BAND_ELEM(J, m-1, n-1)` $= J_{m,n}$. The elements within the band are those with *-mupper* $\le m - n \le$ *mlower*. Alternatively, `BAND_COL(J, j)` returns a pointer to the diagonal element of the `j`-th column of `J`, and if we assign this address to `realtype` `*col_j`, then the `i`-th element of the `j`-th column is given by `BAND_COL_ELEM(col_j, i, j)`, counting from 0. Thus, for (m, n) within the band, $J_{m,n}$ can be loaded by setting `col_n = BAND_COL(J, n-1); BAND_COL_ELEM(col_n, m-1, n-1)` $= J_{m,n}$. The elements of the `j`-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type `DlsMat`. The array `col_n` can be indexed from *-mupper* to *mlower*. For large problems, it is more efficient to use `BAND_COL` and `BAND_COL_ELEM` than to use the `BAND_ELEM` macro. As in the dense case, these macros all number rows and columns starting from 0.

The `DlsMat` type and the accessor macros `BAND_ELEM`, `BAND_COL` and `BAND_COL_ELEM` are documented in the section *Linear Solvers in ARKode*.

If the user's *ARKBandJacFn* function uses difference quotient approximations, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc.. To obtain these, use the ARKodeGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in the header file `sundials_types.h`.

For the sake of uniformity, the arguments *N*, *mlower*, and *mupper* are of type `long int`, even in the case that the LAPACK band solver is to be used.

4.6.10 Jacobian information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG is selected (i.e. ARKSp* is called in step 8 of the section *A skeleton of the user's main program*), the user may provide a function of type `ARKSpilsJacTimesVecFn()` in the following form, to compute matrix-vector products $J * v$. If such a function is not supplied, the default is a difference quotient approximation to these products.

typedef int **(*ARKSpilsJacTimesVecFn)**(N_Vector *v*, N_Vector *Jv*, realtype *t*, N_Vector *y*, N_Vector *fy*, void* *user_data*, N_Vector *tmp*)

This function computes the product $Jv = \left(\frac{\partial f_I}{\partial y}\right) v$ (or an approximation to it).

Arguments:

- *v* – the vector to multiply.
- *Jv* – the output vector computed.
- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f_I(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.
- *tmp* – pointer to memory allocated to a variable of type `N_Vector` which can be used as temporary storage or work space.

Return value: The value to be returned by the Jacobian-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the SPILS generic solver, in which case the integration is halted.

Notes: If the user's *ARKSpilsJacTimesVecFn* function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc.. To obtain these, use the ARKodeGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in the header file `sundials_types.h`.

4.6.11 Preconditioning (linear system solution)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG is selected, and preconditioning is used, then the user must provide a function of type `ARKSpilsPrecSolveFn()` to solve the linear system $Pz = r$, where P may be either a left or right preconditioning matrix. Here P should approximate (at least crudely) the Newton matrix $A = M - \gamma J$, where M is the mass matrix (typically $M = I$ unless working in a finite-element setting) and $J = \frac{\partial f_I}{\partial y}$ If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate A.

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

This function solves the preconditioner system $Pz = r$.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f_I(t, y)$.
- *r* – the right-hand side vector of the linear system.
- *z* – the computed output solution vector.
- *gamma* – the scalar γ appearing in the Newton matrix given by $A = M - \gamma J$.
- *delta* – an input tolerance to be used if an iterative method is employed in the solution. In that case, the resdual vector $Res = r - Pz$ of the system should be made to be less than *delta* in the weighted l_2 norm, i.e. $\left(\sum_{i=1}^n \left(Res_i * ewt_i\right)^2\right)^{1/2} < \delta$, where $\delta =$ *delta*. To obtain the `N_Vector` *ewt*, call `ARKodeGetErrWeights()`.
- *lr* – an input flag indicating whether the preconditioner solve is to use the left preconditioner (*lr* = 1) or the right preconditioner (*lr* = 2).
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.
- *tmp* – pointer to memory allocated to a variable of type `N_Vector` which can be used as temporary storage or work space.

Return value: The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

4.6.12 Preconditioning (Jacobian data)

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

typedef int **`(*ARKSpilsPrecSetupFn)`**(realtype *t*, N_Vector *y*, N_Vector *fy*, booleantype *jok*, booleantype* *jcurPtr*, realtype *gamma*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.

- *fy* – the current value of the vector $f_I(t, y)$.
- *jok* – is an input flag indicating whether the Jacobian-related data needs to be updated. The *jok* argument provides for the reuse of Jacobian data in the preconditioner solve function. When *jok* = `FALSE`, the Jacobian-related data should be recomputed from scratch. When *jok* = `TRUE` 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* = `TRUE` can only occur after a call with *jok* = `FALSE`.
- *jcurPtr* – is a pointer to a flag which should be set to `TRUE` if Jacobian data was recomputed, or set to `FALSE` if Jacobian data was not recomputed, but saved data was still reused.
- *gamma* – the scalar γ appearing in the Newton matrix given by $A = M - \gamma J$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.
- *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type `N_Vector` which can be used as temporary storage or work space.

Return value: The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

Notes: The operations performed by this function might include forming a crude approximate Jacobian, and performing an LU factorization of the resulting approximation to $A = M - \gamma J$.

Each call to the preconditioner setup function is preceded by a call to the implicit `ARKRhsFn()` user function with the same (t, y) arguments. Thus, the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side.

This function is not called in advance of every call to the preconditioner solve function, but rather is called only as often as needed to achieve convergence in the Newton iteration.

If the user's *ARKSpilsPrecSetupFn* function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current step size, the error weights, etc. To obtain these, use the ARKodeGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in the header file `sundials_types.h`.

4.6.13 Mass matrix information (direct method with dense mass matrix)

If the direct linear solver with dense treatment of the mass matrix is used (i.e., `ARKMassDense()` or `ARKMassLapackDense()` is called in Step 10 of the section *A skeleton of the user's main program*), the user may provide a function of type `ARKDlsDenseMassFn()` to provide the mass matrix approximation.

typedef int **(*ARKDlsDenseMassFn)**(long int *N*, realtype *t*, N_Vector *y*, DlsMat *M*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the mass matrix M (or an approximation to it).

Arguments:

- *N* – the size of the ODE system.
- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *M* – the output dense mass matrix (of type `DlsMat`).
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.

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

Return value: An *ARKDlsDenseMassFn* function should return 0 if successful, or a negative value if it failed unrecoverably (in which case the integration is halted, `ARKode()` returns *ARK_MASSSETUP_FAIL* and ARKDENSE sets *last_flag* to *ARKDLS_MASSFUNC_UNRECVR*).

Notes: A user-supplied dense mass matrix function must load the *N* by *N* dense matrix *M* with an approximation to the mass matrix $M(t)$. Only nonzero elements need to be loaded into *M* because it is initialized to the zero matrix before the call to the mass matrix function. The type of *M* is `DlsMat`.

As discussed above in section *Jacobian information (direct method with dense Jacobian)*, the accessor macros `DENSE_ELEM` and `DENSE_COL` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `DlsMat` type. Similarly, the `DlsMat` type and accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in the section *Linear Solvers in ARKode*.

For the sake of uniformity, the argument *N* is of type `long int`, even in the case that the LAPACK dense solver is to be used.

4.6.14 Mass matrix information (direct method with banded mass matrix)

If the direct linear solver with banded treatment of the mass matrix is used (i.e. `ARKMassBand()` or `ARKMassLapackBand()` is called in Step 10 of the section *A skeleton of the user's main program*), the user may provide a function of type `ARKDlsBandMassFn()` to provide the mass matrix approximation.

typedef int **`(*ARKDlsBandMassFn)`**(long int *N*, long int *mupper*, long int *mlower*, realtype *t*, N_Vector *y*, DlsMat *M*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the banded mass matrix M (or an approximation to it).

Arguments:

- *N* – the size of the ODE system.
- *mlower*, *mupper* – the lower and upper half-bandwidths of the mass matrix.
- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector, namely the predicted value of $y(t)$.
- *M* – the output dense mass matrix (of type `DlsMat`).
- *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 ARKDlsBandMassFn as temporary storage or work space.

Return value: An *ARKDlsBandMassFn* function should return 0 if successful, or a negative value if it failed unrecoverably (in which case the integration is halted, `ARKode()` returns *ARK_MASSSETUP_FAIL* and ARKBAND sets *last_flag* to *ARKDLS_MASSFUNC_UNRECVR*).

Notes: A user-supplied banded mass matrix function must load the band matrix *M* of type `DlsMat` with the elements of the mass matrix $M(t)$. Only nonzero elements need to be loaded into *M* because it is initialized to the zero matrix before the call to the mass matrix function.

As discussed above in section *Jacobian information (direct method with banded Jacobian)*, the accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` allow the user to read and write band matrix elements without making specific references to the underlying representation of the `DlsMat` type. Similarly, the `DlsMat` type and the accessor macros `BAND_ELEM`, `BAND_COL` and `BAND_COL_ELEM` are documented in the section *Linear Solvers in ARKode*.

For the sake of uniformity, the arguments *N*, *mlower*, and *mupper* are of type `long int`, even in the case that the LAPACK band solver is to be used.

4.6.15 Mass matrix information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG is selected (i.e. ARKMassSp* is called in step 10 of the section *A skeleton of the user's main program*), the user may provide a function of type `ARKSpilsMassTimesVecFn()` in the following form, to compute matrix-vector products $M * v$.

typedef int **(*ARKSpilsMassTimesVecFn)**(N_Vector *v*, N_Vector *Mv*, realtype *t*, N_Vector *y*, void* *user_data*, N_Vector *tmp*)

This function computes the product $M * v$ (or an approximation to it).

Arguments:

- *v* – the vector to multiply.
- *Mv* – the output vector computed.
- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.
- *tmp* – pointer to memory allocated to a variable of type `N_Vector` which can be used as temporary storage or work space.

Return value: The value to be returned by the mass-matrix-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the SPILS generic solver, in which case the integration is halted.

4.6.16 Mass matrix preconditioning (linear system solution)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG is selected for the mass matrix systems, and preconditioning is used, then the user must provide a function of type `ARKSpilsMassPrecSolveFn()` to solve the linear system $Pz = r$, where P may be either a left or right preconditioning matrix. Here P should approximate (at least crudely) the mass matrix M. If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate M.

typedef int **(*ARKSpilsMassPrecSolveFn)**(realtype *t*, N_Vector *y*, N_Vector *r*, N_Vector *z*, realtype *delta*, int *lr*, void* *user_data*, N_Vector *tmp*)

This function solves the preconditioner system $Pz = r$.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *r* – the right-hand side vector of the linear system.
- *z* – the computed output solution vector.
- *delta* – an input tolerance to be used if an iterative method is employed in the solution. In that case, the resdual vector $Res = r - Pz$ of the system should be made to be less than *delta* in the weighted l_2 norm, i.e. $\left(\sum_{i=1}^n \left(Res_i * ewt_i\right)^2\right)^{1/2} < \delta$, where $\delta = delta$. To obtain the `N_Vector` *ewt*, call `ARKodeGetErrWeights()`.

- *lr* – an input flag indicating whether the preconditioner solve is to use the left preconditioner (*lr* = 1) or the right preconditioner (*lr* = 2).
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.
- *tmp* – pointer to memory allocated to a variable of type `N_Vector` which can be used as temporary storage or work space.

Return value: The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

4.6.17 Mass matrix preconditioning (mass matrix data)

If the user's mass matrix preconditioner requires that any problem data be preprocessed or evaluated, then these actions need to occur within a user-supplied function of type `ARKSpilsMassPrecSetupFn()`.

typedef int **(*ARKSpilsMassPrecSetupFn)**(realtype *t*, N_Vector *y*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function preprocesses and/or evaluates mass-matrix-related data needed by the preconditioner.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *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 as temporary storage or work space.

Return value: The value to be returned by the mass matrix preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

Notes: The operations performed by this function might include forming a mass matrix and performing an incomplete factorization of the result. Although such operations would typically be performed only once at the beginning of a simulation, these may be required if the mass matrix can change as a function of time.

4.6.18 Vector resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatial adaptivity in a PDE simulation), the ARKode integrator may be "resized" between integration steps, through calls to the `ARKodeResize()` function. Typically, when performing adaptive simulations the solution is stored in a customized user-supplied data structure, to enable adaptivity without repeated allocation/deallocation of memory. In these scenarios, it is recommended that the user supply a customized vector kernel to interface between SUNDIALS and their problem-specific data structure. If this vector kernel includes a function to resize a given vector, then this function may be supplied to `ARKodeResize()` so that all internal ARKode vectors may be resized, instead of deleting and re-creating them at each call. This resize function should have the following form:

typedef int **(*ARKVecResizeFn)**(N_Vector *y*, N_Vector *ytemplate*, void* *user_data*)

This function resizes the vector *y* to match the dimensions of the supplied vector, *ytemplate*.

Arguments:

- *y* – the vector to resize.

- *ytemplate* – a vector of the desired size.
- *user_data* – a pointer to user data, the same as the *resize_data* parameter that was passed to `ARKodeResize()`.

Return value: An *ARKVecResizeFn* function should return 0 if it successfully resizes the vector *y*, and a non-zero value otherwise.

Notes: If this function is not supplied, then ARKode will instead destroy the vector *y* and clone a new vector *y* off of *ytemplate*.

4.7 Preconditioner modules

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, ARKode provides two internal preconditioner modules: a banded preconditioner for serial problems (ARKBANDPRE) and a band-block-diagonal preconditioner for parallel problems (ARKBBDPRE).

4.7.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with any of the Krylov iterative linear solvers. It requires that the problem be set up using the NVECTOR_SERIAL module, due to data access patterns. It uses difference quotients of the ODE right-hand side function f_I to generate a band matrix of bandwidth `ml + mu + 1`, where the number of super-diagonals (`mu`, the upper half-bandwidth) and sub-diagonals (`ml`, the lower half-bandwidth) are specified by the user. This band matrix is used to to form a preconditioner the Krylov linear solver. Although this matrix is intended to approximate the Jacobian $J = \frac{\partial f_I}{\partial y}$, it may be a very crude approximation, since the true Jacobian may not be banded, or its true bandwidth may be larger than `ml + mu + 1`. However, as long as the banded approximation generated for the preconditioner is sufficiently accurate, it may speed convergence of the Krylov iteration.

ARKBANDPRE usage

In order to use the ARKBANDPRE module, the user need not define any additional functions. In addition to the header files required for the remainder of the ODE problem (see the section *Access to library and header files*), to use the ARKBANDPRE module, the user's program must include the header file `arkode_bandpre.h` which declares the needed function prototypes. The following is a summary of the usage of this module. Steps that are unchanged from the skeleton program presented in *A skeleton of the user's main program* are *italicized*.

1. *Set problem dimensions*
2. *Set vector of initial values*
3. *Create ARKode object*
4. *Initialize ARKode solver*
5. *Specify integration tolerances*
6. *Set optional inputs*
7. Attach iterative linear solver module, one of:
   - `ier = ARKSpgmr(...);`
   - `ier = ARKSpbcg(...);`
   - `ier = ARKSptfqmr(...);`

- `ier = ARKSpfgmr(...);`
- `ier = ARKPcg(...);`

8. 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.

9. *Set linear solver optional inputs*

   Note that the user should not overwrite the preconditioner setup function or solve function through calls to the ARKSpilsSet* optional input functions.

10. *Specify rootfinding problem*
11. *Advance solution in time*
12. Get optional outputs

    Additional optional outputs associated with ARKBANDPRE are available by way of the two routines described below, `ARKBandPrecGetWorkSpace()` and `ARKBandPrecGetNumRhsEvals()`.

13. *Free solver memory*
14. *Deallocate memory for solution vector*

We note that at present, the ARKBANDPRE preconditioner may not be used for problems involving a non-identity mass matrix, $M \neq I$, although support for this is planned for the near future.

ARKBANDPRE user-callable functions

The ARKBANDPRE preconditioner module is initialized and attached by calling the following function:

int **ARKBandPrecInit**(void* *arkode_mem*, long int *N*, long int *mu*, long int *ml*)

Initializes the ARKBANDPRE preconditioner and allocates required (internal) memory for it.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *N* – problem dimension (size of ODE system).
- *mu* – upper half-bandwidth of the Jacobian approximation.
- *ml* – lower half-bandwidth of the Jacobian approximation.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_ILL_INPUT* if an input has an illegal value
- *ARKSPILS_MEM_FAIL* if a memory allocation request failed

Notes: The banded approximate Jacobian will have nonzero elements only in locations (i, j) with $ml \leq j - i \leq mu$.

The following two optional output functions are available for use with the ARKBANDPRE module:

int **`ARKBandPrecGetWorkSpace`**(void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)

Returns the sizes of the ARKBANDPRE real and integer workspaces.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lenrwLS* – the number of `realtype` values in the ARKBANDPRE workspace.
- *leniwLS* – the number of integer values in the ARKBANDPRE workspace.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: In terms of the problem size N and $smu = \min(N-1, mu{+}ml)$, the actual size of the real workspace is $(2\,ml + mu + smu + 2)N$ `realtype` words, and the actual size of the integer workspace is N integer words.

The workspaces referred to here exist in addition to those given by the corresponding function `ARKSpilsGetWorkspace()`.

int **`ARKBandPrecGetNumRhsEvals`**(void* *arkode_mem*, long int* *nfevalsBP*)

Returns the number of calls made to the user-supplied right-hand side function f_I for constructing the finite-difference banded Jacobian approximation used within the preconditioner setup function.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *nfevalsBP* – number of calls to f_I

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: The counter *nfevalsBP* is distinct from the counter *nfevalsLS* returned by the corresponding function `ARKSpilsGetNumRhsEvals()` and also from *nfi_evals* returned by `ARKodeGetNumRhsEvals()`. The total number of right-hand side function evaluations is the sum of all three of these counters, plus the *nfe_evals* counter for f_E calls returned by `ARKodeGetNumRhsEvals()`.

4.7.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver (such as ARKode) lies in the solution of partial differential equations (PDEs). Moreover, Krylov iterative methods are used on many such problems due to the nature of the underlying linear system of equations that needs to solved at each time step. For many PDEs, the linear algebraic system is large, sparse and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner is required. Otherwise, the rate of convergence of the Krylov iterative method is usually slow, and degrades as the PDE mesh is refined. Typically, an effective preconditioner must be problem-specific. However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used with CVODE for several realistic, large-scale problems [HT1998] and is included in a software module within the ARKode package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers. It generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called ARKBBDPRE.

One way to envision these preconditioners is to think of the computational PDE domain as being subdivided into Q non-overlapping subdomains, where each subdomain is assigned to one of the Q MPI tasks used to solve the ODE system. The basic idea is to isolate the preconditioning so that it is local to each process, and also to use a (possibly cheaper) approximate right-hand side function for construction of this preconditioning matrix. This requires the definition of a new function $g(t,y) \approx f_I(t,y)$ that will be used to construct the BBD preconditioner matrix. As with the rest of ARKode, we assume here that the ODE system is written as

$$M\dot{y} = f_E(t,y) + f_I(t,y),$$

where f_I corresponds to the ODE components to be treated implicitly. The user may set $g = f_I$, if no less expensive approximation is desired.

Corresponding to the domain decomposition, there is a decomposition of the solution vector y into Q disjoint blocks y_q, and a decomposition of g into blocks g_q. The block g_q depends both on y_p and on components of blocks $y_{q'}$ associated with neighboring subdomains (so-called ghost-cell data). If we let $\bar{y}_q$ denote y_q augmented with those other components on which g_q depends, then we have

$$g(t,y) = \left[g_1(t,\bar{y}_1), g_2(t,\bar{y}_2), \ldots, g_Q(t,\bar{y}_Q)\right]^T,$$

and each of the blocks $g_q(t,\bar{y}_q)$ is decoupled from one another.

The preconditioner associated with this decomposition has the form

$$P = \text{diag}[P_1, P_2, \ldots, P_Q]$$

where

$$P_q \approx M - \gamma J_q$$

and where J_q is a difference quotient approximation to $\frac{\partial g_q}{\partial \bar{y}_q}$. This matrix is taken to be banded, with upper and lower half-bandwidths *mudq* and *mldq* defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using *mudq* + *mldq* + 2 evaluations of g_m, but only a matrix of bandwidth *mukeep* + *mlkeep* + 1 is retained. Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of g, if smaller values provide a more efficient preconditioner. The solution of the complete linear system

$$Px = b$$

reduces to solving each of the distinct equations

$$P_q x_q = b_q, \quad q = 1, \ldots, Q,$$

and this is done by banded LU factorization of P_q followed by a banded backsolve.

Similar block-diagonal preconditioners could be considered with different treatments of the blocks P_q. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

ARKBBDPRE user-supplied functions

The ARKBBDPRE module calls two user-provided functions to construct P: a required function *gloc* (of type `ARKLocalFn()`) which approximates the right-hand side function $g(t,y) \approx f_I(t,y)$ and which is computed locally, and an optional function *cfn* (of type `ARKCommFn()`) which performs all interprocess communication necessary to evaluate the approximate right-hand side g. These are in addition to the user-supplied right-hand side function f_I. Both functions take as input the same pointer *user_data* that is passed by the user to `ARKodeSetUserData()` and that was passed to the user's function f_I. The user is responsible for providing space (presumably within *user_data*) for components of y that are communicated between processes by *cfn*, and that are then used by *gloc*, which should not do any communication.

typedef int **(*ARKLocalFn)**(long int *Nlocal*, realtype *t*, N_Vector *y*, N_Vector *glocal*, void* *user_data*)

This *gloc* function computes $g(t,y)$. It fills the vector *glocal* as a function of *t* and *y*.

Arguments:

- *Nlocal* – the local vector length
- *t* – the value of the independent variable
- *y* – the value of the dependent variable vector on this process
- *glocal* – the output vector of $g(t,y)$ on this process
- *user_data* – a pointer to user data, the same as the *user_data* parameter passed to `ARKodeSetUserData()`.

Return value: An *ARKLocalFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKode will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `ARKode()` will return *ARK_LSETUP_FAIL*).

Notes: This function should assume that all interprocess communication of data needed to calculate *glocal* has already been done, and that this data is accessible within user data.

The case where g is mathematically identical to f_I is allowed.

typedef int **(*ARKCommFn)**(long int *Nlocal*, realtype *t*, N_Vector *y*, void* *user_data*)

This *cfn* function performs all interprocess communication necessary for the executation of the *gloc* function above, using the input vector *y*.

Arguments:

- *Nlocal* – the local vector length
- *t* – the value of the independent variable
- *y* – the value of the dependent variable vector on this process
- *user_data* – a pointer to user data, the same as the *user_data* parameter passed to `ARKodeSetUserData()`.

Return value: An *ARKCommFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKode will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `ARKode()` will return *ARK_LSETUP_FAIL*).

Notes: The *cfn* function is expected to save communicated data in space defined within the data structure *user_data*.

Each call to the *cfn* function is preceded by a call to the right-hand side function f_I with the same (t,y) arguments. Thus, *cfn* can omit any communication done by f_I if relevant to the evaluation of *glocal*. If all necessary communication was done in f_I, then *cfn* = `NULL` can be passed in the call to `ARKBBDPrecInit()` (see below).

ARKBBDPRE usage

In addition to the header files required for the integration of the ODE problem (see the section *Access to library and header files*), to use the ARKBBDPRE module, the user's program must include the header file `arkode_bbdpre.h` which declares the needed function prototypes.

The following is a summary of the proper usage of this module. Steps that are unchanged from the skeleton program presented in *A skeleton of the user's main program* are *italicized*.

1. *Initialize MPI*
2. *Set problem dimensions*
3. *Set vector of initial values*
4. *Create ARKode object*
5. *Initialize ARKode solver*
6. *Specify integration tolerances*
7. *Set optional inputs*
8. Attach iterative linear solver, one of:
   - `ier = ARKSpgmr(...);`
   - `ier = ARKSpbcg(...);`
   - `ier = ARKSptfqmr(...);`
   - `ier = ARKSpfgmr(...);`
   - `ier = ARKPcg(...);`
9. Initialize the ARKBBDPRE preconditioner module

   Specify the upper and lower half-bandwidths for computation `mudq` and `mldq`, the upper and lower half-bandwidths for storage `mukeep` and `mlkeep`, and call

   `ier = ARKBBDPrecInit(arkode_mem, Nlocal, mudq, mldq, mukeep, mlkeep, dqrely, gloc, cfn);`

   to allocate memory and initialize the internal preconditioner data. The last two arguments of `ARKBBDPrecInit()` are the two user-supplied functions of type `ARKLocalFn()` and `ARKCommFn()` described above, respectivelyl.
10. *Set the linear solver optional inputs*

    Note that the user should not overwrite the preconditioner setup function or solve function through calls to ARKSPILS optional input functions.
11. *Specify rootfinding problem*
12. *Advance solution in time*
13. *Get optional outputs*

    Additional optional outputs associated with ARKBBDPRE are available through the routines `ARKBBDPrecGetWorkSpace()` and `ARKBBDPrecGetNumGfnEvals()`.
14. *Free solver memory*
15. *Deallocate memory for solution vector*
16. *Finalize MPI*

We note that at present, the ARKBBDPRE preconditioner may not be used for problems involving a non-identity mass matrix, $M \neq I$, although support for this is planned for the near future.

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*, long int *Nlocal*, long int *mudq*, long int *mldq*, long int *mukeep*, long int *mlkeep*, realtype *dqrely*, ARKLocalFn *gloc*, ARKCommFn *cfn*)

Initializes and allocates (internal) memory for the ARKBBDPRE preconditioner.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *Nlocal* – local vector length.
- *mudq* – upper half-bandwidth to be used in the difference quotient Jacobian approximation.
- *mldq* – lower half-bandwidth to be used in the difference quotient Jacobian approximation.
- *mukeep* – upper half-bandwidth of the retained banded approximate Jacobian block.
- *mlkeep* – lower half-bandwidth of the retained banded approximate Jacobian block.
- *dqrely* – the relative increment in components of *y* used in the difference quotient approximations. The default is *dqrely* = $\sqrt{\text{unit roundoff}}$, which can be specified by passing *dqrely* = 0.0.
- *gloc* – the name of the C function (of type `ARKLocalFn()`) which computes the approximation $g(t, y) \approx f_I(t, y)$.
- *cfn* – the name of the C function (of type `ARKCommFn()`) which performs all interprocess communication required for the computation of $g(t, y)$.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_ILL_INPUT* if an input has an illegal value
- *ARKSPILS_MEM_FAIL* if a memory allocation request failed

Notes: If one of the half-bandwidths *mudq* or *mldq* to be used in the difference quotient calculation of the approximate Jacobian is negative or exceeds the value *Nlocal*-1, it is replaced by 0 or *Nlocal*-1 accordingly.

The half-bandwidths *mudq* and *mldq* need not be the true half-bandwidths of the Jacobian of the local block of g when smaller values may provide a greater efficiency.

Also, the half-bandwidths *mukeep* and *mlkeep* of the retained banded approximate Jacobian block may be even smaller than *mudq* and *mldq*, to reduce storage and computational costs further.

For all four half-bandwidths, the values need not be the same on every processor.

The ARKBBDPRE module also provides a reinitialization function to allow solving a sequence of problems of the same size, with the same linear solver choice, provided there is no change in *Nlocal*, *mukeep*, or *mlkeep*. After solving one problem, and after calling `ARKodeReInit()` to re-initialize ARKode for a subsequent problem, a call to `ARKBBDPrecReInit()` can be made to change any of the following: the half-bandwidths *mudq* and *mldq* used in the difference-quotient Jacobian approximations, the relative increment *dqrely*, or one of the user-supplied functions *gloc* and *cfn*. If there is a change in any of the linear solver inputs, an additional call to `ARKSpgmr()`, `ARKSpbcg()`, `ARKSptfqmr()`, `ARKSpfgmr()`, or `ARKPcg()`, and/or one or more of the corresponding ARKSpilsSet* functions, must also be made (in the proper order).

int **`ARKBBDPrecReInit`** (void* *arkode_mem*, long int *mudq*, long int *mldq*, realtype *dqrely*)
Re-initializes the ARKBBDPRE preconditioner module.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *mudq* – upper half-bandwidth to be used in the difference quotient Jacobian approximation.
- *mldq* – lower half-bandwidth to be used in the difference quotient Jacobian approximation.
- *dqrely* – the relative increment in components of *y* used in the difference quotient approximations. The default is $dqrely = \sqrt{\text{unit roundoff}}$, which can be specified by passing *dqrely* = 0.0.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: If one of the half-bandwidths *mudq* or *mldq* is negative or exceeds the value *Nlocal*-1, it is replaced by 0 or *Nlocal*-1 accordingly.

The following two optional output functions are available for use with the ARKBBDPRE module:

int **`ARKBBDPrecGetWorkSpace`** (void* *arkode_mem*, long int* *lenrwBBDP*, long int* *leniwBBDP*)
Returns the processor-local ARKBBDPRE real and integer workspace sizes.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *lenrwBBDP* – the number of `realtype` values in the ARKBBDPRE workspace.
- *leniwBBDP* – the number of integer values in the ARKBBDPRE workspace.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: In terms of *Nlocal* and *smu = min(Nlocal-1, mukeep+mlkeep)*, the actual size of the real workspace is *(2 mlkeep + mukeep + smu + 2)*Nlocal* `realtype` words, and the actual size of the integer workspace is *Nlocal* integer words. These values are local to each process.

The workspaces referred to here exist in addition to those given by the corresponding function `ARKSpilsGetWorkSpace()`.

int **`ARKBBDPrecGetNumGfnEvals`** (void* *arkode_mem*, long int* *ngevalsBBDP*)
Returns the number of calls made to the user-supplied *gloc* function (of type `ARKLocalFn()`) due to the finite difference approximation of the Jacobian blocks used within the preconditioner setup function.

Arguments:

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

- *ngevalsBBDP* – the number of calls made to the user-supplied *gloc* function.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_PMEM_NULL* if the preconditioner memory is `NULL`

In addition to the *ngevalsBBDP gloc* evaluations, the costs associated with ARKBBDPRE also include *nlinsetups* LU factorizations, *nlinsetups* calls to *cfn*, *npsolves* banded backsolve calls, and *nfevalsLS* right-hand side function evaluations, where *nlinsetups* is an optional ARKode output and *npsolves* and *nfevalsLS* are linear solver optional outputs (see the table *Iterative linear solvers optional output functions*).

CHAPTER

FIVE

FARKODE, AN INTERFACE MODULE FOR FORTRAN APPLICATIONS

The FARKODE interface module is a package of C functions which support the use of the ARKODE solver for the solution of ODE systems

$$M\dot{y} = f_E(t,y) + f_I(t,y),$$

in a mixed Fortran/C setting. While ARKODE is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in Fortran. This package provides the necessary interface to ARKODE for both the serial and the parallel NVECTOR implementations.

5.1 Important notes on portability

In this package, the names of the interface functions, and the names of the Fortran user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files `farkode.h`, `farkroot.h`, `farkbp.h`, and `farkbbd.h`. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h`. The mapping defined by `F77_FUNC` in turn transforms the C interface names to match the name-mangling approach used by the supplied Fortran compiler.

By "name-mangling", we mean that due to the case-independent nature of the Fortran language, Fortran compilers convert all subroutine and object names to use either all lower-case or all upper-case characters, and append either zero, one or two underscores as a prefix or suffix the the name. For example, the Fortran subroutine `MyFunction()` will be changed to one of `myfunction`, `MYFUNCTION`, `myfunction__`, `MYFUNCTION_`, and so on, depending on the Fortran compiler used.

SUNDIALS determines this name-mangling scheme at configuration time (see *ARKode Installation Procedure*).

5.2 Fortran Data Types

Throughout this documentation, we will refer to data types according to their usage in C. The equivalent types to these may vary, depending on your computer architecture and on how SUNDIALS was compiled (see *ARKode Installation Procedure*). A Fortran user should first determine the equivalent types for their architecture and compiler, and then take care that all arguments passed through this Fortran/C interface are declared of the appropriate type.

Integers: SUNDIALS uses both `int` and `long int` types:

- `int` – equivalent to an `INTEGER` or `INTEGER*4` in Fortran
- `long int` – this will depend on the computer architecture:
  - 32-bit architecture – equivalent to an `INTEGER` or `INTEGER*4` in Fortran
  - 64-bit architecture – equivalent to an `INTEGER*8` in Fortran

Real numbers: As discussed in *ARKode Installation Procedure*, at compilation SUNDIALS allows the configuration option `--with-precision`, that accepts values of `single`, `double` or `extended` (the default is `double`). This choice dictates the size of a `realtype` variable. The corresponding Fortran types for these `realtype` sizes are:

- `single` – equivalent to a `REAL` or `REAL*4` in Fortran
- `double` – equivalent to a `DOUBLE PRECISION` or `REAL*8` in Fortran
- `extended` – equivalent to a `REAL*16` in Fortran

Details on the Fortran interface to ARKode are provided in the following sub-sections:

5.2.1 FARKODE routines

In this section, we list the full set of user-callable functions comprising the FARKODE solver interface. For each function, we list the corresponding ARKode functions, to provide a mapping between the two solver interfaces. Further documentation on each FARKODE function is provided in the following sections, *Usage of the FARKODE interface module*, *FARKODE optional output*, *Usage of the FARKROOT interface to rootfinding* and *Usage of the FARKODE interface to built-in preconditioners*. Additionally, all Fortran and C functions below are hyperlinked to their definitions in the documentation, for simplified access.

Interface to the NVECTOR modules

- `FNVINITS()` (defined by NVECTOR_SERIAL) interfaces to `N_VNewEmpty_Serial()`.
- `FNVINITP()` (defined by NVECTOR_PARALLEL) interfaces to `N_VNewEmpty_Parallel()`.

Interface to the main ARKODE module

- `FARKMALLOC()` interfaces to `ARKodeCreate()`, `ARKodeSetUserData()`, and `ARKodeInit()`, as well as one of `ARKodeSStolerances()` or `ARKodeSVtolerances()`.
- `FARKREINIT()` interfaces to `ARKodeReInit()`.
- `FARKRESIZE()` interfaces to `ARKodeResize()`.
- `FARKSETIIN()` and `FARKSETRIN()` interface to the ARKodeSet* functions (see *Optional input functions*).
- `FARKEWTSET()` interfaces to `ARKodeWFtolerances()`.
- `FARKADAPTSET()` interfaces to `ARKodeSetAdaptivityFn()`.
- `FARKEXPSTABSET()` interfaces to `ARKodeSetStabilityFn()`.
- `FARKODE()` interfaces to `ARKode()`, the ARKodeGet* functions (see *Optional output functions*), and to the optional output functions for the selected linear solver module (see *Optional output functions*).
- `FARKDKY()` interfaces to the interpolated output function `ARKodeGetDky()`.
- `FARKGETERRWEIGHTS()` interfaces to `ARKodeGetErrWeights()`.
- `FARKGETESTLOCALERR()` interfaces to `ARKodeGetEstLocalErrors()`.
- `FARKFREE()` interfaces to `ARKodeFree()`.

Interface to the linear solver modules

- `FARKDENSE()` interfaces to `ARKDense()`.
- `FARKLAPACKDENSE()` interfaces to `ARKLapackDense()`.
- `FARKDENSESETJAC()` interfaces to `ARKDlsSetDenseJacFn()`.
- `FARKBAND()` interfaces to `ARKBand()`.
- `FARKLAPACKBAND()` interfaces to `ARKLapackBand()`.
- `FARKBANDSETJAC()` interfaces to `ARKDlsSetBandJacFn()`.
- `FARKSPGMR()` interfaces to `ARKSpgmr()` and the SPGMR optional input functions (see *Table: Optional inputs for ARKSPILS*).
- `FARKSPGMRREINIT()` interfaces to the SPGMR optional input functions (see *Table: Optional inputs for ARKSPILS*).
- `FARKSPBCG()` interfaces to `ARKSpbcg()` and the SPBCG optional input functions (see *Table: Optional inputs for ARKSPILS*).
- `FARKSPBCGREINIT()` interfaces to the SPBCG optional input functions.
- `FARKSPTFQMR()` interfaces to `ARKSptfqmr()` and the SPTFQMR optional input functions.
- `FARKSPTFQMRREINIT()` interfaces to the SPTFQMR optional input functions.
- `FARKSPFGMR()` interfaces to `ARKSpfgmr()` and the SPFGMR optional input functions (see *Table: Optional inputs for ARKSPILS*).
- `FARKSPFGMRREINIT()` interfaces to the SPFGMR optional input functions (see *Table: Optional inputs for ARKSPILS*).
- `FARKPCG()` interfaces to `ARKPcg()` and the PCG optional input functions (see *Table: Optional inputs for ARKSPILS*).
- `FARKPCGREINIT()` interfaces to the PCG optional input functions.
- `FARKSPILSSETJAC()` interfaces to `ARKSpilsSetJacTimesVecFn()`.
- `FARKSPILSSETPREC()` interfaces to `ARKSpilsSetPreconditioner()`.

User-supplied routines

As with the native C interface, the FARKode solver interface requires user-supplied functions to specify the ODE problem to be solved. In contrast to the case of direct use of ARKode, and of most Fortran ODE solvers, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. As a result, whether using a purely implicit, purely explicit, or mixed implicit-explicit solver, routines for both $f_E(t, y)$ and $f_I(t, y)$ must be provided by the user (though either of which may do nothing):

FARKODE routine (FORTRAN, user-supplied)	ARKode interface function type
`FARKIFUN()`	`ARKRhsFn()`
`FARKEFUN()`	`ARKRhsFn()`

In addition, as with the native C interface a user may provide additional routines to assist in the solution process. Each of the following user-supplied routines is activated by calling the specified "activation" routine:

FARKODE routine (FORTRAN, user-supplied)	ARKode interface function type	FARKODE "activation" routine
`FARKDJAC()`	`ARKDlsDenseJacFn()`	`FARKDENSESETJAC()`
`FARKBJAC()`	`ARKDlsBandJacFn()`	`FARKBANDSETJAC()`
`FARKPSET()`	`ARKSpilsPrecSetupFn()`	`FARKSPILSSETPREC()`
`FARKPSOL()`	`ARKSpilsPrecSolveFn()`	`FARKSPILSSETPREC()`
`FARKJTIMES()`	`ARKSpilsJacTimesVecFn()`	`FARKSPILSSETJAC()`
`FARKEWT()`	`ARKEwtFn()`	`FARKEWTSET()`
`FARKADAPT()`	`ARKAdaptFn()`	`FARKADAPTSET()`
`FARKEXPSTAB()`	`ARKExpStabFn()`	`FARKEXPSTABSET()`

5.2.2 Usage of the FARKODE interface module

The usage of FARKODE requires calls to five or more interface functions, depending on the method options selected, and two or more user-supplied routines which define the problem to be solved. These function calls and user routines are summarized individually below. Some details on specific argument options, and the user is referred to the description of the corresponding C interface ARKode functions for complete information on the arguments of any given user-callable interface routine. The usage of FARKODE for rootfinding and with preconditioner modules is described in later subsections.

We note that at present, support for non-identity mass matrices, $M \neq I$ is only provided in the C/C++ interface to ARKode, although support within the Fortran interface is planned for the near future.

In the instructions below, steps marked [**S**] apply to the serial NVECTOR implementation (NVECTOR_SERIAL) only, while those marked with a [**P**] apply to NVECTOR_PARALLEL.

Right-hand side specification

The user must in all cases supply the following Fortran routines:

subroutine **`FARKIFUN`** (*T*, *Y*, *YDOT*, *IPAR*, *RPAR*, *IER*)

Sets the *YDOT* array to $f_I(t, y)$, the implicit portion of the right-hand side of the ODE system, as function of the independent variable $T = t$ and the array of dependent state variables $Y = y$.

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing state variables.
- *YDOT* (`realtype`, output) – array containing state derivatives.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 success, >0 recoverable error, <0 unrecoverable error).

subroutine **`FARKEFUN`** (*T*, *Y*, *YDOT*, *IPAR*, *RPAR*, *IER*)

Sets the *YDOT* array to $f_E(t, y)$, the explicit portion of the right-hand side of the ODE system, as function of the independent variable $T = t$ and the array of dependent state variables $Y = y$.

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing state variables.
- *YDOT* (`realtype`, output) – array containing state derivatives.

- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 success, >0 recoverable error, <0 unrecoverable error).

For purely explicit problems, although the routine `FARKIFUN()` must exist, it will never be called, and may remain empty. Similarly, for purely implicit problems, `FARKEFUN()` will never be called and must exist and may remain empty.

NVECTOR module initialization

[**S**] To initialize the serial NVECTOR module, the user must call the following function with the argument *KEY* = 4.

subroutine FNVINITS (*KEY*, *NEQ*, *IER*)
Initializes the Fortran interface to the serial NVECTOR module.

Arguments:

- *KEY* (`int`, input) – integer flag denoting which solver is to be used (1 is CVODE, 2 is IDA, 3 is KINSOL and 4 is ARKode).
- *NEQ* (`long int`, input) – size of the ODE system.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

[**P**] To initialize the parallel NVECTOR module, the user must call the following function with the argument *KEY* = 4.

subroutine FNVINITP (*COMM*, *KEY*, *NLOCAL*, *NGLOBAL*, *IER*)
Initializes the Fortran interface to the parallel NVECTOR module.

Arguments:

- *COMM* (`int`, input) – the MPI communicator.
- *KEY* (`int`, input) – integer flag denoting which solver is to be used (1 is CVODE, 2 is IDA, 3 is KINSOL and 4 is ARKode).
- *NLOCAL* (`long int`, input) – local vector size on this processor.
- *NGLOBAL* (`long int`, input) – the size of the ODE system, and the global size of vectors (the sum of all values of *NLOCAL*).
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Notes: If the header file `sundials_config.h` defines `SUNDIALS_MPI_COMM_F2C` to be 1 (meaning the MPI implementation used to build SUNDIALS includes the `MPI_Comm_f2c()` function), then COMM can be any valid MPI communicator. Otherwise, `MPI_COMM_WORLD` will be used, so the user can just pass an integer value as a placeholder.

Problem specification

To set various problem and solution parameters and allocate internal memory, the user must call `FARKMALLOC()`.

subroutine FARKMALLOC (*T0*, *Y0*, *IMEX*, *IATOL*, *RTOL*, *ATOL*, *IOUT*, *ROUT*, *IPAR*, *RPAR*, *IER*)
Initializes the Fortran interface to the ARKode solver, providing interfaces to the C routines `ARKodeCreate()`, `ARKodeSetUserData()`, and `ARKodeInit()`, as well as one of `ARKodeSStolerances()` or `ARKodeSVtolerances()`.

Arguments:

- *T0* (`realtype`, input) – initial value of t.

- *Y0* (`realtype`, input) – array of initial conditions.
- *IMEX* (`int`, input) – flag denoting basic integration method: 0 = implicit, 1 = explicit, 2 = ImEx.
- *IATOL* (`int`, input) – type for absolute tolerance input *ATOL*: 1 = scalar, 2 = array, 3 = user-supplied function; the user must subsequently call `FARKEWTSET()` and supply a routine `FARKEWT()` to compute the error weight vector.
- *RTOL* (`realtype`, input) – scalar relative tolerance.
- *ATOL* (`realtype`, input) – scalar or array absolute tolerance.
- *IOUT* (`long int`, input/output) – array of length 22 for integer optional outputs.
- *ROUT* (`realtype`, input/output) – array of length 6 for real optional outputs.
- *IPAR* (`long int`, input/output) – array of user integer data, which will be passed unmodified to all user-provided routines.
- *RPAR* (`realtype`, input/output) – array with user real data, which will be passed unmodified to all user-provided routines.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Notes: Modifications to the user data arrays *IPAR* and *RPAR* inside a user-provided routine will be propagated to all subsequent calls to such routines. The optional outputs associated with the main ARKode integrator are listed in *Table: Optional FARKODE integer outputs* and *Table: Optional FARKODE real outputs*, in the section *FARKODE optional output*.

As an alternative to providing tolerances in the call to `FARKMALLOC()`, the user may provide a routine to compute the error weights used in the WRMS norm evaluations. If supplied, it must have the following form:

subroutine FARKEWT (*Y*, *EWT*, *IPAR*, *RPAR*, *IER*)

It must set the positive components of the error weight vector *EWT* for the calculation of the WRMS norm of *Y*.

Arguments:

- *Y* (`realtype`, input) – array containing state variables.
- *EWT* (`realtype`, output) – array containing the error weight vector.
- *IPAR* (`long int`, input) – array containing the integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing the real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

If the `FARKEWT()` routine is provided, then, following the call to `FARKMALLOC()`, the user must call the function `FARKEWTSET()`.

subroutine FARKEWTSET (*FLAG*, *IER*)

Informs FARKODE to use the user-supplied `FARKEWT()` function.

Arguments:

- *FLAG* (`int`, input) – flag, use "1" to denoting to use `FARKEWT()`.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Setting optional inputs

Unlike ARKode's C interface, that provides separate functions for setting each optional input, FARKODE uses only two functions, that accept keywords to specify which optional input should be set to the provided value. These routines are `FARKSETIIN()` and `FARKSETRIN()`, and are further described below.

subroutine FARKSETIIN(*KEY*, *IVAL*, *IER*)

Specification routine to pass optional integer inputs to the `FARKODE()` solver.

Arguments:

- *KEY* (quoted string, input) – which optional input is set (see *Table: Keys for setting FARKODE integer optional inputs*).
- *IVAL* (`long int`, input) – the integer input value to be used.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Table: Keys for setting FARKODE integer optional inputs

Key	ARKode routine
`ORDER`	`ARKodeSetOrder()`
`DENSE_ORDER`	`ARKodeSetDenseOrder()`
`LINEAR`	`ARKodeSetLinear()`
`NONLINEAR`	`ARKodeSetNonlinear()`
`EXPLICIT`	`ARKodeSetExplicit()`
`IMPLICIT`	`ARKodeSetImplicit()`
`IMEX`	`ARKodeSetImEx()`
`IRK_TABLE_NUM`	`ARKodeSetIRKTableNum()`
`ERK_TABLE_NUM`	`ARKodeSetERKTableNum()`
`ARK_TABLE_NUM` *(a)*	`ARKodeSetARKTableNum()`
`MAX_NSTEPS`	`ARKodeSetMaxNumSteps()`
`HNIL_WARNS`	`ARKodeSetMaxHnilWarns()`
`PREDICT_METHOD`	`ARKodeSetPredictorMethod()`
`MAX_ERRFAIL`	`ARKodeSetMaxErrTestFails()`
`MAX_CONVFAIL`	`ARKodeSetMaxConvFails()`
`MAX_NITERS`	`ARKodeSetMaxNonlinIters()`
`ADAPT_SMALL_NEF`	`ARKodeSetSmallNumEFails()`
`LSETUP_MSBP`	`ARKodeSetMaxStepsBetweenLSet()`

(a) When setting `ARK_TABLE_NUM`, pass in *IVAL* as an array of length 2, specifying the IRK table number first, then the ERK table number.

subroutine FARKSETRIN(*KEY*, *RVAL*, *IER*)

Specification routine to pass optional real inputs to the `FARKODE()` solver.

Arguments:

- *KEY* (quoted string, input) – which optional input is set (see *Table: Keys for setting FARKODE real optional inputs*).
- *RVAL* (`realtype`, input) – the real input value to be used.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Table: Keys for setting FARKODE real optional inputs

Key	ARKode routine
`INIT_STEP`	`ARKodeSetInitStep()`
`MAX_STEP`	`ARKodeSetMaxStep()`
`MIN_STEP`	`ARKodeSetMinStep()`
`STOP_TIME`	`ARKodeSetStopTime()`
`NLCONV_COEF`	`ARKodeSetNonlinConvCoef()`
`ADAPT_CFL`	`ARKodeSetCFLFraction()`
`ADAPT_SAFETY`	`ARKodeSetSafetyFactor()`
`ADAPT_BIAS`	`ARKodeSetErrorBias()`
`ADAPT_GROWTH`	`ARKodeSetMaxGrowth()`
`ADAPT_ETAMX1`	`ARKodeSetMaxFirstGrowth()`
`ADAPT_BOUNDS`	`ARKodeSetFixedStepBounds()`
`ADAPT_ETAMXF`	`ARKodeSetMaxEFailGrowth()`
`ADAPT_ETACF`	`ARKodeSetMaxCFailGrowth()`
`NEWT_CRDOWN`	`ARKodeSetNonlinCRDown()`
`NEWT_RDIV`	`ARKodeSetNonlinRDiv()`
`LSETUP_DGMAX`	`ARKodeSetDeltaGammaMax()`

If a user wishes to reset all of the options to their default values, they may call the routine `FARKSETDEFAULTS()`.

subroutine FARKSETDEFAULTS (*IER*)

Specification routine to reset all FARKODE optional inputs to their default values.

Arguments:

- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Optional advanced FARKODE inputs

FARKODE supplies additional routines to specify optional advanced inputs to the `ARKode()` solver. These are summarized below, and the user is referred to their C routine counterparts for more complete information.

subroutine FARKSETERKTABLE (*S*, *Q*, *P*, *C*, *A*, *B*, *BEMBED*, *IER*)

Interface to the routine `ARKodeSetERKTable()`.

Arguments:

- *S* (`int`, input) – number of stages in the table.
- *Q* (`int`, input) – global order of accuracy of the method.
- *P* (`int`, input) – global order of accuracy of the embedding.
- *C* (`realtype`, input) – array of length *S* containing the stage times.
- *A* (`realtype`, input) – array of length *S*S* containing the ERK coefficients (stored in row-major, “C”, order).
- *B* (`realtype`, input) – array of length *S* containing the solution coefficients.
- *BEMBED* (`realtype`, input) – array of length *S* containing the embedding coefficients.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

subroutine FARKSETIRKTABLE (*S*, *Q*, *P*, *C*, *A*, *B*, *BEMBED*, *IER*)

Interface to the routine `ARKodeSetIRKTable()`.

Arguments:

- *S* (`int`, input) – number of stages in the table.
- *Q* (`int`, input) – global order of accuracy of the method.
- *P* (`int`, input) – global order of accuracy of the embedding.
- *C* (`realtype`, input) – array of length *S* containing the stage times.
- *A* (`realtype`, input) – array of length *S*S* containing the IRK coefficients (stored in row-major, "C", order).
- *B* (`realtype`, input) – array of length *S* containing the solution coefficients.
- *BEMBED* (`realtype`, input) – array of length *S* containing the embedding coefficients.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

subroutine FARKSETARKTABLES (*S*, *Q*, *P*, *C*, *AI*, *AE*, *B*, *BEMBED*, *IER*)
Interface to the routine `ARKodeSetARKTables()`.

Arguments:

- *S* (`int`, input) – number of stages in the table.
- *Q* (`int`, input) – global order of accuracy of the method.
- *P* (`int`, input) – global order of accuracy of the embedding.
- *C* (`realtype`, input) – array of length *S* containing the stage times.
- *AI* (`realtype`, input) – array of length *S*S* containing the IRK coefficients (stored in row-major, "C", order)
- *AE* (`realtype`, input) – array of length *S*S* containing the ERK coefficients (stored in row-major, "C", order)
- *B* (`realtype`, input) – array of length *S* containing the solution coefficients
- *BEMBED* (`realtype`, input) – array of length *S* containing the embedding coefficients
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure)

Additionally, a user may set the accuracy-based step size adaptivity strategy (and it's associated parameters) through a call to `FARKSETADAPTIVITYMETHOD()`, as described below.

subroutine FARKSETADAPTIVITYMETHOD (*IMETHOD*, *IDEFAULT*, *IPQ*, *PARAMS*, *IER*)
Specification routine to set the step size adaptivity strategy and parameters within the `FARKODE()` solver. Interfaces with the C routine `ARKodeSetAdaptivityMethod()`.

Arguments:

- *IMETHOD* (`int`, input) – choice of adaptivity method.
- *IDEFAULT* (`int`, input) – flag denoting whether to use default parameters (1) or that customized parameters will be supplied (1).
- *IPQ* (`int`, input) – flag denoting whether to use the embedding order of accuracy (0) or the method order of accuracy (1) within step adaptivity algorithm.
- *PARAMS* (`realtype`, input) – array of 3 parameters to be used within the adaptivity strategy.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Lastly, the user may provide functions to aid/replace those within ARKode for handling adaptive error control and explicit stability. The former of these is designed for advanced users who wish to investigate custom step adaptivity approaches as opposed to using any of those built-in to ARKode. In ARKode's C/C++ interface, this would be provided by a function of type `ARKAdaptFn()`; in the Fortran interface this is provided through the user-supplied function:

subroutine FARKADAPT(*Y*, *T*, *H1*, *H2*, *H3*, *E1*, *E2*, *E3*, *Q*, *P*, *HNEW*, *IPAR*, *RPAR*, *IER*)
It must set the new step size *HNEW* based on the three previous steps (*H1*, *H2*, *H3*) and the three previous error estimates (*E1*, *E2*, *E3*).

Arguments:

- *Y* (`realtype`, input) – array containing state variables.
- *T* (`realtype`, input) – current value of the independent variable.
- *H1* (`realtype`, input) – current step size.
- *H2* (`realtype`, input) – previous step size.
- *H3* (`realtype`, input) – previous-previous step size.
- *E1* (`realtype`, input) – estimated temporal error in current step.
- *E2* (`realtype`, input) – estimated temporal error in previous step.
- *E3* (`realtype`, input) – estimated temporal error in previous-previous step.
- *Q* (`int`, input) – global order of accuracy for RK method.
- *P* (`int`, input) – global order of accuracy for RK embedding.
- *HNEW* (`realtype`, output) – array containing the error weight vector.
- *IPAR* (`long int`, input) – array containing the integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing the real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

This routine is enabled by a call to the activation routine:

subroutine FARKADAPTSET(*FLAG*, *IER*)
Informs FARKODE to use the user-supplied `FARKADAPT()` function.

Arguments:

- *FLAG* (`int`, input) – flag, use "1" to denoting to use `FARKADAPT()`, or use "0" to denote a return to the default adaptivity strategy.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Note: The call to `FARKADAPTSET()` must occur *after* the call to `FARKMALLOC()`.

Similarly, if either an explicit or mixed implicit-explicit integration method is to be employed, the user may specify a function to provide the maximum explicitly-stable step for their problem. Again, in the C/C++ interface this would be a function of type `ARKExpStabFn()`, while in ARKode's Fortran interface this must be given through the user-supplied function:

subroutine FARKEXPSTAB(*Y*, *T*, *HSTAB*, *IPAR*, *RPAR*, *IER*)
It must set the maximum explicitly-stable step size, *HSTAB*, based on the current solution, *Y*.

Arguments:

- *Y* (`realtype`, input) – array containing state variables.
- *T* (`realtype`, input) – current value of the independent variable.
- *HSTAB* (`realtype`, output) – maximum explicitly-stable step size.
- *IPAR* (`long int`, input) – array containing the integer user data that was passed to `FARKMALLOC()`.

- *RPAR* (`realtype`, input) – array containing the real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

This routine is enabled by a call to the activation routine:

subroutine FARKEXPSTABSET (*FLAG*, *IER*)

Informs FARKODE to use the user-supplied `FARKEXPSTAB()` function.

Arguments:

- *FLAG* (`int`, input) – flag, use "1" to denoting to use `FARKEXPSTAB()`, or use "0" to denote a return to the default error-based stability strategy.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Note: The call to `FARKEXPSTABSET()` must occur *after* the call to `FARKMALLOC()`.

Linear solver specification

In the case of using either an implicit or ImEx method, the solution of each Runge-Kutta stage may involve the solution of linear systems related to the Jacobian $J = \frac{\partial f_I}{\partial y}$ of the implicit portion of the ODE system. ARKode presently includes seven choices for the treatment of these systems, and the user of FARKODE must call a routine with a specific name to make the desired choice.

[S] Dense treatment of the linear system

To use the direct dense linear solver based on the internal SUNDIALS implementation, the user must call the `FARKDENSE()` routine:

subroutine FARKDENSE (*NEQ*, *IER*)

Interfaces with the `ARKDense()` function to specify use of the dense direct linear solver.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

Alteratively, to use the LAPACK-based direct dense linear solver, a user must call the similar `FARKLAPACKDENSE()` routine:

subroutine FARKLAPACKDENSE (*NEQ*, *IER*)

Interfaces with the `ARKLapackDense()` function to specify use of the LAPACK the dense direct linear solver.

Arguments:

- *NEQ* (`int`, input) – size of the ODE system.
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

As an option when using either of these dense linear solvers, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \frac{\partial f_I}{\partial y}$. If supplied, it must have the following form:

subroutine FARKDJAC (*NEQ*, *T*, *Y*, *FY*, *DJAC*, *H*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)

Interface to provide a user-supplied dense Jacobian approximation function (of type `ARKDenseJacFn()`), to be used by the `FARKDENSE()` solver.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing values of the dependent state variables.
- *FY* (`realtype`, input) – array containing values of the dependent state derivatives.
- *DJAC* (`realtype` of size (NEQ,NEQ), output) – 2D array containing the Jacobian entries.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *WK1*, *WK2*, *WK3* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: Typically this routine will use only *NEQ*, *T*, *Y*, and *DJAC*. It must compute the Jacobian and store it column-wise in *DJAC*.

If the above routine uses difference quotient approximations, it may need to access the error weight array *EWT* in the calculation of suitable increments. The array *EWT* can be obtained by calling `FARKGETERRWEIGHTS()` using one of the work arrays as temporary storage for *EWT*. It may also need the unit roundoff, which can be obtained as the optional output *ROUT(6)*, passed from the calling program to this routine using either *RPAR* or a common block.

If the `FARKDJAC()` routine is provided, then, following the call to `FARKDENSE()` or `FARKLAPACKDENSE()`, the user must call the routine `FARKDENSESETJAC()`:

subroutine `FARKDENSESETJAC` (*FLAG*, *IER*)

Interface to the `ARKDenseSetJacFn()` function, specifying to use the user-supplied routine `FARKDJAC()` for the Jacobian approximation.

Arguments:

- *FLAG* (`int`, input) – any nonzero value specifies to use `FARKDJAC()`.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error occurred).

[S] Band treatment of the linear system

To use the direct band linear solver that is based on the internal SUNDIALS implementation, the user must call the `FARKBAND()` routine.

subroutine `FARKBAND` (*NEQ*, *MU*, *ML*, *IER*)

Interfaces with the `ARKBand()` function to specify use of the dense banded linear solver.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *MU* (`long int`, input) – upper half-bandwidth.
- *ML* (`long int`, input) – lower half-bandwidth.
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

Alteratively, to use the LAPACK-based direct banded linear solver, a user must call the similar `FARKLAPACKBAND()` routine:

subroutine FARKLAPACKBAND (*NEQ*, *MU*, *ML*, *IER*)

Interfaces with the `ARKLapackBand()` function to specify use of the dense banded linear solver.

Arguments:

- *NEQ* (`int`, input) – size of the ODE system.
- *MU* (`int`, input) – upper half-bandwidth.
- *ML* (`int`, input) – lower half-bandwidth.
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

As an option when using either of these banded linear solvers, the user may supply a routine that computes a banded approximation of the linear system Jacobian $J = \frac{\partial f_I}{\partial y}$. If supplied, it must have the following form:

subroutine FARKBJAC (*NEQ*, *MU*, *ML*, *MDIM*, *T*, *Y*, *FY*, *BJAC*, *H*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)

Interface to provide a user-supplied band Jacobian approximation function (of type `ARKBandJacFn()`), to be used by the `FARKBAND()` solver.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *MU* (`long int`, input) – upper half-bandwidth.
- *ML* (`long int`, input) – lower half-bandwidth.
- *MDIM* (`long int`, input) – leading dimension of *BJAC* array.
- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing dependent state variables.
- *FY* (`realtype`, input) – array containing dependent state derivatives.
- *BJAC* (`realtype` of size *(MDIM,NEQ)*, output) – 2D array containing the Jacobian entries.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *WK1*, *WK2*, *WK3* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: Typically this routine will use only *NEQ*, *MU*, *ML*, *T*, *Y*, and *BJAC*. It must load the *MDIM* by *N* array *BJAC* with the Jacobian matrix at the current (t, y) in band form. Store in *BJAC(k,j)* the Jacobian element $J_{i,j}$ with *k = i - j + MU + 1* (or *k = 1, ..., ML+MU+1*) and *j = 1, ..., N*.

If the above routine uses difference quotient approximations, it may need to use the error weight array *EWT* in the calculation of suitable increments. The array *EWT* can be obtained by calling `FARKGETERRWEIGHTS()` using one of the work arrays as temporary storage for *EWT*. It may also need the unit roundoff, which can be obtained as the optional output *ROUT(6)*, passed from the calling program to this routine using either *RPAR* or a common block.

If the `FARKBJAC()` routine is provided, then, following the call to either `FARKBAND()` or `FARKLAPACKBAND()`, the user must call the routine `FARKBANDSETJAC()`.

subroutine FARKBANDSETJAC (*FLAG*, *IER*)

Interface to the `ARKBandSetJacFn()` function, specifying to use the user-supplied routine `FARKBJAC()` for the Jacobian approximation.

Arguments:

- *FLAG* (`int`, input) – any nonzero value specifies to use `FARKBJAC()`.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error occurred).

[S][P] SPGMR treatment of the linear systems

For the Scaled Preconditioned GMRES solution of the linear systems, the user must call the `FARKSPGMR()` routine:

subroutine FARKSPGMR(*IPRETYPE*, *IGSTYPE*, *MAXL*, *DELT*, *IER*)

Interfaces with the `ARKSpgmr()` and ARKSpilsSet* routines to specify use of the SPGMR iterative linear solver.

Arguments:

- *IPRETYPE* (`int`, input) – preconditioner type: 0 = none, 1 = left only, 2 = right only, 3 = both sides.
- *IGSTYPE* (`int`, input) – Gram-schmidt orthogonalization process: 1 = modified G-S, 2 = classical G-S.
- *MAXL* (`int`; input) – maximum Krylov subspace dimension (0 for default).
- *DELT* (`realtype`, input) – linear convergence tolerance factor (0.0 for default).
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

For descriptions of the optional user-supplied routines for use with `FARKSPGMR()` see the section *[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR/SPFGMR/PCG*.

[S][P] SPBCG treatment of the linear systems

For the Scaled Preconditioned Bi-CGStab solution of the linear systems, the user must call the `FARKSPBCG()` routine:

subroutine FARKSPBCG(*IPRETYPE*, *MAXL*, *DELT*, *IER*)

Interfaces with the `ARKSpbcg()` and ARKSpilsSet* routines to specify use of the SPBCG iterative linear solver.

Arguments: The arguments are the same as those with the same names for `FARKSPGMR()`.

For descriptions of the optional user-supplied routines for use with `FARKSPBCG()` see the section *[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR/SPFGMR/PCG*.

[S][P] SPTFQMR treatment of the linear systems

For the Scaled Preconditioned TFQMR solution of the linear systems, the user must call the `FARKSPTFQMR()` routine:

subroutine FARKSPTFQMR(*IPRETYPE*, *MAXL*, *DELT*, *IER*)

Interfaces with the `ARKSptfqmr()` and ARKSpilsSet* routines to specify use of the SPTFQMR iterative linear solver.

Arguments: The arguments are the same as those with the same names for `FARKSPGMR()`.

For descriptions of the optional user-supplied routines for use with `FARKSPTFQMR()` see the next section.

[S][P] SPFGMR treatment of the linear systems

For the Scaled Preconditioned Flexible Generalized Minimum Residual solution of the linear systems, the user must call the `FARKSPFGMR()` routine:

subroutine FARKSPFGMR(*IPRETYPE*, *IGSTYPE*, *MAXL*, *DELT*, *IER*)

Interfaces with the `ARKSpfgmr()` and ARKSpilsSet* routines to specify use of the SPFGMR iterative linear solver.

Arguments: The arguments are the same as those for `FARKSPGMR()`.

For descriptions of the optional user-supplied routines for use with `FARKSPFGMR()` see the section *[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR/SPFGMR/PCG*.

[S][P] PCG treatment of the linear systems

For the Preconditioned Conjugate Gradient solution of symmetric linear systems, the user must call the `FARKPCG()` routine:

subroutine FARKPCG(*IPRETYPE*, *MAXL*, *DELT*, *IER*)

Interfaces with the `ARKPcg()` and ARKSpilsSet* routines to specify use of the PCG iterative linear solver.

Arguments: The arguments are the same as those with the same names for `FARKSPGMR()`.

For descriptions of the optional user-supplied routines for use with `FARKPCG()` see the section *[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR/SPFGMR/PCG*.

[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR/SPFGMR/PCG

With treatment of the linear systems by any of the Krylov iterative solvers, there are three optional user-supplied routines – `FARKJTIMES()`, `FARKPSET()` and `FARKPSOL()`. The specifications of these functions are given below.

The first of these optional routines when using a Krylov iterative solver is a routine to compute the product of the system Jacobian $J = \frac{\partial f_I}{\partial y}$ and a given vector v. If supplied, it must have the following form:

subroutine FARKJTIMES(*V*, *FJV*, *T*, *Y*, *FY*, *H*, *IPAR*, *RPAR*, *WORK*, *IER*)

Interface to provide a user-supplied Jacobian-times-vector product approximation function (corresponding to a C interface routine of type `ARKSpilsJacTimesVecFn()`), to be used by one of the Krylov iterative linear solvers.

Arguments:

- *V* (`realtype`, input) – array containing the vector to multiply.
- *FJV* (`realtype`, output) – array containing resulting product vector.
- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing dependent state variables.
- *FY* (`realtype`, input) – array containing dependent state derivatives.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *WORK* (`realtype`, input) – array containing temporary workspace of same size as *Y*.

- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: Typically this routine will use only *NEQ*, *T*, *Y*, *V*, and *FJV*. It must compute the product vector Jv, where v is given in *V*, and the product is stored in *FJV*.

If this routine has been supplied by the user, then, following the call to `FARKSPGMR()`, `FARKSPBCG()`, `FARKSPTFQMR()`, `FARKSPFGMR()` or `FARKPCG()`, the user must call the routine `FARKSPILSSETJAC()` with *FLAG* $\neq 0$ to specify use of the user-supplied Jacobian-times-vector function:

subroutine FARKSPILSSETJAC (*FLAG*, *IER*)

Interface to the function `ARKSpilsSetJacTimesVecFn()` to specify use of the user-supplied Jacobian-times-vector function `FARKJTIMES()`.

Arguments:

- *FLAG* (`int`, input) – flag denoting to use `FARKJTIMES()` routine.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

If preconditioning is to be performed during the Krylov solver (i.e. the solver was set up with *IPRETYPE* $\neq 0$), then the user must also call the routine `FARKSPILSSETPREC()` with *FLAG* $\neq 0$:

subroutine FARKSPILSSETPREC (*FLAG*, *IER*)

Interface to the function `ARKSpilsSetPreconditioner()` to specify use of the user-supplied preconditioner setup and solve functions, `FARKPSET()` and `FARKPSOL()`, respectively.

Arguments:

- *FLAG* (`int`, input) – flag denoting use of user-supplied preconditioning routines.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

In addition, the user must provide the following two routines to implement the preconditioner setup and solve functions to be used within the solve.

subroutine FARKPSET (*T*, *Y*, *FY*, *JOK*, *JCUR*, *GAMMA*, *H*, *IPAR*, *RPAR*, *V1*, *V2*, *V3*, *IER*)

User-supplied preconditioner setup routine (of type `ARKSpilsPrecSetupFn()`).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – current dependent state variable array.
- *FY* (`realtype`, input) – current dependent state variable derivative array.
- *JOK* (`int`, input) – flag indicating whether Jacobian-related data needs to be recomputed: 0 = recompute, 1 = reuse with the current value of *GAMMA*.
- *JCUR* (`realtype`, output) – return flag to denote if Jacobian data was recomputed (1=yes, 0=no).
- *GAMMA* (`realtype`, input) – Jacobian scaling factor.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to `FARKMALLOC()`.
- *V1*, *V2*, *V3* (`realtype`, input) – arrays containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable failure, <0 if a non-recoverable failure).

Notes: This routine must set up the preconditioner P to be used in the subsequent call to `FARKPSOL()`. The preconditioner (or the product of the left and right preconditioners if using both) should be an approximation to the matrix $M - \gamma J$, where M is the system mass matrix, γ is the input *GAMMA*, and $J = \frac{\partial f_I}{\partial y}$.

subroutine `FARKPSOL` (*T*, *Y*, *FY*, *R*, *Z*, *GAMMA*, *DELTA*, *LR*, *IPAR*, *RPAR*, *VT*, *IER*)

User-supplied preconditioner solve routine (of type `ARKSpilsPrecSolveFn()`).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – current dependent state variable array.
- *FY* (`realtype`, input) – current dependent state variable derivative array.
- *R* (`realtype`, input) – right-hand side array.
- *Z* (`realtype`, output) – solution array.
- *GAMMA* (`realtype`, input) – Jacobian scaling factor.
- *DELTA* (`realtype`, input) – desired residual tolerance.
- *LR* (`int`, input) – flag denoting to solve the right or left preconditioner system: 1 = left preconditioner, 2 = right preconditioner.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to `FARKMALLOC()`.
- *VT* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable failure, <0 if a non-recoverable failure).

Notes: Typically this routine will use only *NEQ*, *T*, *Y*, *GAMMA*, *R*, *LR*, and *Z*. It must solve the preconditioner linear system $Pz = r$. The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the matrix $M - \gamma J$, where M is the system mass matrix, γ is the input GAMMA, and $J = \frac{\partial f_I}{\partial y}$.

Notes:

1. If the user's `FARKJTIMES()` or `FARKPSET()` routine uses difference quotient approximations, it may need to use the error weight array *EWT* and/or the unit roundoff, in the calculation of suitable increments. Also, if `FARKPSOL()` uses an iterative method in its solution, the residual vector $\rho = r - Pz$ of the system should be made less than $\delta =$ *DELTA* in the weighted l2 norm, i.e.

$$\left(\sum_i \left(\rho_i \, EWT_i\right)^2\right)^{1/2} < \delta.$$

2. If needed in `FARKJTIMES()`, `FARKPSOL()`, or `FARKPSET()`, the error weight array *EWT* can be obtained by calling `FARKGETERRWEIGHTS()` using one of the work arrays as temporary storage for *EWT*.
3. If needed in `FARKJTIMES()`, `FARKPSOL()`, or `FARKPSET()`, the unit roundoff can be obtained as the optional output *ROUT(6)* (available after the call to `FARKMALLOC()`) and can be passed using either the *RPAR* user data array or a common block.

Problem solution

Carrying out the integration is accomplished by making calls to `FARKODE()`.

subroutine FARKODE (*TOUT*, *T*, *Y*, *ITASK*, *IER*)

Fortran interface to the C routine `ARKode()` for performing the solve, along with many of the ARK*Get* routines for reporting on solver statistics.

Arguments:

- *TOUT* (`realtype`, input) – next value of t at which a solution is desired.
- *T* (`realtype`, output) – current value of independent variable reached by the solver.
- *Y* (`realtype`, output) – array containing dependent state variables on output.
- *ITASK* (`int`, input) – task indicator :
  - 1 = normal mode (overshoot *TOUT* and interpolate)
  - 2 = one-step mode (return after each internal step taken)
  - 3 = normal 'tstop' mode (like 1, but integration never proceeds past *TSTOP*, which must be specified through a preceding call to `FARKSETRIN()` using the key *STOP_TIME*)
  - 4 = one step 'tstop' mode (like 2, but integration never goes past *TSTOP*).
- *IER* (int, output) – completion flag:
  - 0 = success,
  - 1 = tstop return,
  - 2 = root return,
  - values -1, ..., -10 are failure modes (see `ARKode()` and *Appendix: ARKode Constants*).

Notes: The current values of the optional outputs are immediately available in *IOUT* and *ROUT* upon return from this function (see *Table: Optional FARKODE integer outputs* and *Table: Optional FARKODE real outputs*).

Additional solution output

After a successful return from `FARKODE()`, the routine `FARKDKY()` may be used to obtain a derivative of the solution, of order up to 3, at any t within the last step taken.

subroutine FARKDKY (*T*, *K*, *DKY*, *IER*)

Fortran interface to the C routine `ARKDKY()` for interpolating output of the solution or its derivatives at any point within the last step taken.

Arguments:

- *T* (`realtype`, input) – time at which solution derivative is desired, within the interval $[t_n - h, t_n]$.
- *K* (`int`, input) – derivative order $(0 \le k \le 3)$.
- *DKY* (`realtype`, output) – array containing the computed *K*-th derivative of y.
- *IER* (`int`, output) – return flag (0 if success, <0 if an illegal argument).

Problem reinitialization

To re-initialize the ARKode solver for the solution of a new problem of the same size as one already solved, the user must call `FARKREINIT()`:

subroutine **`FARKREINIT`**(*T0*, *Y0*, *IMEX*, *IATOL*, *RTOL*, *ATOL*, *IER*)

Re-initializes the Fortran interface to the ARKode solver.

Arguments: The arguments have the same names and meanings as those of `FARKMALLOC()`.

Notes: This routine performs no memory allocation, instead using the existing memory created by the previous `FARKMALLOC()` call. The call to specify the linear system solution method may or may not be needed.

Following a call to `FARKREINIT()`, a call to specify the linear system solver must be made if the choice of linear solver is being changed. Otherwise, a call to reinitialize the linear solver last used is only needed if linear solver input parameters need modification.

In the case of the BAND solver, for any change in the half-bandwidth parameters, call `FARKBAND()` (or `FARKLAPACKBAND()`) again, as described above.

In the case of SPGMR, for a change of inputs other than *MAXL*, the user may call the routine `FARKSPGMRREINIT()` to reinitialize SPGMR without reallocating its memory, as follows:

subroutine **`FARKSPGMRREINIT`**(*IPRETYPE*, *IGSTYPE*, *DELT*, *IER*)

Re-initializes the Fortran interface to the SPGMR linear solver.

Arguments: The arguments have the same names and meanings as those of `FARKSPGMR()`.

However, if *MAXL* is being changed, then the user should call `FARKSPGMR()` instead, since memory will need to be deallocated/reallocated by the solver.

In the case of SPBCG, for a change in any inputs, the user can reinitialize SPBCG without reallocating its memory by calling `FARKSPBCGREINIT()`, as follows:

subroutine **`FARKSPBCGREINIT`**(*IPRETYPE*, *MAXL*, *DELT*, *IER*)

Re-initializes the Fortran interface to the SPBCG linear solver.

Arguments: The arguments have the same names and meanings as those of `FARKSPBCG()`.

In the case of SPTFQMR, for a change in any inputs, the user can reinitialize SPTFQMR without reallocating its memory by calling `FARKSPTFQMRREINIT()`, as follows:

subroutine **`FARKSPTFQMRREINIT`**(*IPRETYPE*, *MAXL*, *DELT*, *IER*)

Re-initializes the Fortran interface to the SPBTFQMR linear solver.

Arguments: The arguments have the same names and meanings as those of `FARKSPTFQMR()`.

In the case of SPFGMR, for a change of inputs other than *MAXL*, the user may call the routine `FARKSPFGMRREINIT()` to reinitialize SPFGMR without reallocating its memory, as follows:

subroutine **`FARKSPFGMRREINIT`**(*IPRETYPE*, *IGSTYPE*, *DELT*, *IER*)

Re-initializes the Fortran interface to the SPFGMR linear solver.

Arguments: The arguments have the same names and meanings as those of `FARKSPFGMR()`.

However, if *MAXL* is being changed, then the user should call `FARKSPFGMR()` instead, since memory will need to be deallocated/reallocated by the solver.

In the case of PCG, for a change in any inputs, the user can reinitialize PCG without reallocating its memory by calling `FARKPCGREINIT()`, as follows:

subroutine **`FARKPCGREINIT`**(*IPRETYPE*, *MAXL*, *DELT*, *IER*)

Re-initializes the Fortran interface to the PCG linear solver.

Arguments: The arguments have the same names and meanings as those of `FARKPCG()`.

Resizing the ODE system

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when solving a spatially-adaptive PDE), the `FARKODE()` integrator may be "resized" between integration steps, through calls to the `FARKRESIZE()` function, that interfaces with the C routine `ARKodeResize()`. This function modifies ARKode's internal memory structures to use the new problem size, without destruction of the temporal adaptivity heuristics. It is assumed that the dynamical time scales before and after the vector resize will be comparable, so that all time-stepping heuristics prior to calling `FARKRESIZE()` remain valid after the call. If instead the dynamics should be re-calibrated, the FARKODE memory structure should be deleted with a call to `FARKFREE()`, and re-created with a call to `FARKMALLOC()`.

subroutine FARKRESIZE(*T0*, *Y0*, *HSCALE*, *ITOL*, *RTOL*, *ATOL*, *IER*)

Re-initializes the Fortran interface to the ARKode solver for a differently-sized ODE system.

Arguments:

- *T0* (`realtype`, input) – initial value of the independent variable t.
- *Y0* (`realtype`, input) – array of dependent-variable initial conditions.
- *HSCALE* (`realtype`, input) – desired step size scale factor:
  - 1.0 is the default,
  - any value <= 0.0 results in the default.
- *ITOL* (`int`, input) – flag denoting that a new relative tolerance and vector of absolute tolerances are supplied in the *RTOL* and *ATOL* arguments:
  - 0 = retain the current scalar-valued relative and absolute tolerances, or the user-supplied error weight function, `FARKEWT()`.
  - 1 = *RTOL* contains the new scalar-valued relative tolerance and *ATOL* contains a new array of absolute tolerances.
- *RTOL* (`realtype`, input) – scalar relative tolerance.
- *ATOL* (`realtype`, input) – array of absolute tolerances.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Notes: This routine performs the opposite set of of operations as `FARKREINIT()`: it does not reinitialize any of the time-step heuristics, but it does perform memory reallocation.

Following a call to `FARKRESIZE()`, a call to specify the linear system solver must be made **after** the call to `FARKRESIZE()`, since the internal data structures for the linear solver will also be the incorrect size.

If any user-supplied linear solver helper routines were used (Jacobian evaluation, Jacobian-vector product, preconditioning, etc.), then the relevant "set" routines to specify their usage must be called again **following** the re-specification of the linear solver module.

Memory deallocation

To free the internal memory created by `FARKMALLOC()`, the user may call `FARKFREE()`, as follows:

subroutine FARKFREE()

Frees the internal memory created by `FARKMALLOC()`.

Arguments: None.

5.2.3 FARKODE optional output

We note that the optional inputs to FARKODE have already been described in the section *Setting optional inputs*.

IOUT and ROUT arrays

In the Fortran interface, the optional outputs from the `FARKODE()` solver are accessed not through individual functions, but rather through a pair of user-allocated arrays, *IOUT* (having `long int` type) of dimension at least 22, and *ROUT* (having `realtype` type) of dimension at least 6. These arrays must be allocated by the user program that calls `FARKODE()`, that passes them through the Fortran interface as arguments to `FARKMALLOC()`. Following this call, `FARKODE()` will modify the entries of these arrays to contain all optional output values provided to a Fortran user.

In the following tables, *Table: Optional FARKODE integer outputs* and *Table: Optional FARKODE real outputs*, we list the entries in these arrays by index, naming them according to their role with the main ARKode solver, and list the relevant ARKode C/C++ function that is actually called to extract the output value. Similarly, optional integer output values that are specific to the ARKDENSE and ARKBAND linear solvers are listed in *Table: Optional ARKDENSE and ARKBAND outputs*, while integer optional output values specific to the ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG iterative linear solvers are listed in *Table: Optional ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG outputs*.

For more details on the optional inputs and outputs to ARKode, see the sections *Optional input functions* and *Optional output functions*.

Table: Optional FARKODE integer outputs

IOUT Index	Optional output	ARKode function
1	LENRW	`ARKodeGetWorkSpace()`
2	LENIW	`ARKodeGetWorkSpace()`
3	NST	`ARKodeGetNumSteps()`
4	NST_STB	`ARKodeGetNumExpSteps()`
5	NST_ACC	`ARKodeGetNumAccSteps()`
6	NST_ATT	`ARKodeGetNumStepAttempts()`
7	NFE	`ARKodeGetNumRhsEvals()` (num f_E calls)
8	NFI	`ARKodeGetNumRhsEvals()` (num f_I calls)
9	NSETUPS	`ARKodeGetNumLinSolvSetups()`
10	NETF	`ARKodeGetNumErrTestFails()`
11	NNI	`ARKodeGetNumNonlinSolvIters()`
12	NCFN	`ARKodeGetNumNonlinSolvConvFails()`
13	NGE	`ARKodeGetNumGEvals()`

Table: Optional FARKODE real outputs

ROUT Index	Optional output	ARKode function
1	H0U	`ARKodeGetActualInitStep()`
2	HU	`ARKodeGetLastStep()`
3	HCUR	`ARKodeGetCurrentStep()`
4	TCUR	`ARKodeGetCurrentTime()`
5	TOLSF	`ARKodeGetTolScaleFactor()`
6	UROUND	`UNIT_ROUNDOFF` (see the section *Data Types*)

Table: Optional ARKDENSE and ARKBAND outputs

IOUT Index	Optional output	ARKode function
14	LENRWLS	`ARKDlsGetWorkSpace()`
15	LENIWLS	`ARKDlsGetWorkSpace()`
16	LSTF	`ARKDlsGetLastFlag()`
17	NFELS	`ARKDlsGetNumRhsEvals()`
18	NJE	`ARKDlsGetNumJacEvals()`

Table: Optional ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG outputs

IOUT Index	Optional output	ARKode function
14	LENRWLS	`ARKSpilsGetWorkSpace()`
15	LENIWLS	`ARKSpilsGetWorkSpace()`
16	LSTF	`ARKSpilsGetLastFlag()`
17	NFELS	`ARKSpilsGetNumRhsEvals()`
18	NJTV	`ARKSpilsGetNumJtimesEvals()`
19	NPE	`ARKSpilsGetNumPrecEvals()`
20	NPS	`ARKSpilsGetNumPrecSolves()`
21	NLI	`ARKSpilsGetNumLinIters()`
22	NCFL	`ARKSpilsGetNumConvFails()`

Additional optional output routines

In addition to the optional inputs communicated through FARKSET* calls and the optional outputs extracted from *IOUT* and *ROUT*, the following user-callable routines are available.

To obtain the error weight array *EWT*, containing the multiplicative error weights used in the WRMS norms, the user may call the routine `FARKGETERRWEIGHTS()` as follows:

subroutine FARKGETERRWEIGHTS(*EWT*, *IER*)

Retrieves the current error weight vector (interfaces with `ARKodeGetErrWeights()`).

Arguments:

- *EWT* (`realtype`, output) – array containing the error weight vector.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: The array *EWT* must have already been allocated by the user, of the same size as the solution array *Y*.

Similarly, to obtain the estimated local truncation errors, following a successful call to `FARKODE()`, the user may call the routine `FARKGETESTLOCALERR()` as follows:

subroutine FARKGETESTLOCALERR(*ELE*, *IER*)

Retrieves the current local truncation error estimate vector (interfaces with `ARKodeGetEstLocalErrors()`).

Arguments:

- *ELE* (`realtype`, output) – array with the estimated local truncation error vector.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: The array *ELE* must have already been allocated by the user, of the same size as the solution array *Y*.

5.2.4 Usage of the FARKROOT interface to rootfinding

The FARKROOT interface package allows programs written in Fortran to use the rootfinding feature of the ARKode solver module. The user-callable functions in FARKROOT, with the corresponding ARKODE functions, are as follows:

- `FARKROOTINIT()` interfaces to `ARKodeRootInit()`,
- `FARKROOTINFO()` interfaces to `ARKodeGetRootInfo()`, and
- `FARKROOTFREE()` interfaces to `ARKodeRootInit()`, freeing memory by calling the initializer with no root functions.

Note that at this time, FARKROOT does not provide support to specify the direction of zero-crossing that is to be monitored. Instead, all roots are considered. However, the actual direction of zero-crossing may be captured by the user through monitoring the sign of any non-zero elements in the array *INFO* returned by `FARKROOTINFO()`.

In order to use the rootfinding feature of ARKode, after calling `FARKMALLOC()` but prior to calling `FARKODE()`, the user must call `FARKROOTINIT()` to allocate and initialize memory for the FARKROOT module:

subroutine `FARKROOTINIT`(*NRTFN*, *IER*)
Initializes the Fortran interface to the FARKROOT module.

Arguments:

- *NRTFN* (`int`, input) – total number of root functions.
- *IER* (`int`, output) – return flag (0 success, -1 if ARKode memory is `NULL`, and -11 if a memory allocation error occurred).

If rootfinding is enabled, the user must specify the functions whose roots are to be found. These rootfinding functions should be implemented in the user-supplied `FARKROOTFN()` subroutine:

subroutine `FARKROOTFN`(*T*, *Y*, *G*, *IPAR*, *RPAR*, *IER*)
User supplied function implementing the vector-valued function $g(t,y)$ such that the roots of the *NRTFN* components $g_i(t,y)=0$ are sought.

Arguments:

- *T* (`realtype`, input) – independent variable value t.
- *Y* (`realtype`, input) – dependent variable array y.
- *G* (`realtype`, output) – function value array $g(t,y)$.
- *IPAR* (`long int`, input/output) – integer user data array, the same as the array passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input/output) – real-valued user data array, the same as the array passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 success, <0 if error).

When making calls to `FARKODE()` to solve the ODE system, the occurrence of a root is flagged by the return value *IER = 2*. In that case, if *NRTFN > 1*, the functions $g_i(t,y)$ which were found to have a root can be identified by calling the routine `FARKROOTINFO()`:

subroutine `FARKROOTINFO`(*NRTFN*, *INFO*, *IER*)
Initializes the Fortran interface to the FARKROOT module.

Arguments:

- *NRTFN* (`int`, input) – total number of root functions.

- *INFO* (`int`, input/output) – array of length *NRTFN* with root information (must be allocated by the user). For each index, *i = 1, ..., NRTFN*:
  - *INFO(i) = 1* if $g_i(t, y)$ was found to have a root, and g_i is increasing.
  - *INFO(i) = -1* if $g_i(t, y)$ was found to have a root, and g_i is decreasing.
  - *INFO(i) = 0* otherwise.
- *IER* (`int`, output) – return flag (0 success, < 0 if error).

The total number of calls made to the root function `FARKROOTFN()`, denoted *NGE*, can be obtained from *IOUT(12)*. If the FARKODE/ARKode memory block is reinitialized to solve a different problem via a call to `FARKREINIT()`, then the counter *NGE* is reset to zero.

Lastly, to free the memory resources allocated by a prior call to `FARKROOTINIT()`, the user must make a call to `FARKROOTFREE()`:

subroutine `FARKROOTFREE`()

Frees memory associated with the FARKODE rootfinding module.

5.2.5 Usage of the FARKODE interface to built-in preconditioners

The FARKODE interface enables usage of the two built-in preconditioning modules ARKBANDPRE and ARKBBD-PRE. Details on how these preconditioners work are provided in the section *Preconditioner modules*. In this section, we focus specifically on the Fortran interface to these modules.

Usage of the FARKBP interface to ARKBANDPRE

The FARKBP interface module is a package of C functions which, as part of the FARKODE interface module, support the use of the ARKode solver with the serial NVECTOR_SERIAL module, and the combination of the ARKBAND-PRE preconditioner module (see the section *A serial banded preconditioner module*) with any of the Krylov iterative linear solvers.

The two user-callable functions in this package, with the corresponding ARKode function around which they wrap, are:

- `FARKBPINIT()` interfaces to `ARKBandPrecInit()`.
- `FARKBPOPT()` interfaces to the ARKBANDPRE optional output functions, `ARKBandPrecGetWorkSpace()` and `ARKBandPrecGetNumRhsEvals()`.

As with the rest of the FARKODE routines, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `farkbp.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in the section *Usage of the FARKODE interface module* are *italicized*.

1. *Right-hand side specification*
2. *NVECTOR module initialization*
3. *Problem specification*
4. *Set optional inputs*
5. Linear solver specification

   First, specify one of the ARKSPILS iterative linear solvers, by calling one of `FARKSPGMR()`, `FARKSPBCG()`, `FARKSPTFQMR()`, `FARKSPFGMR()` or `FARKPCG()`.

Optionally, to specify that SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG should use the supplied `FARKJTIMES()` routine, the user should call `FARKSPILSSETJAC()` with FLAG $\neq 0$, as described in the section *[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR/SPFGMR/PCG*.

Then, to initialize the ARKBANDPRE preconditioner, call the routine `FARKBPINIT()`, as follows:

subroutine FARKBPINIT(*NEQ*, *MU*, *ML*, *IER*)

Interfaces with the `ARKBandPrecInit()` function to allocate memory and initialize data associated with the ARKBANDPRE preconditioner.

Arguments:

- *NEQ* (`long int`, input) – problem size.
- *MU* (`long int`, input) – upper half-bandwidth of the band matrix that is retained as an approximation of the Jacobian.
- *ML* (`long int`, input) – lower half-bandwidth of the band matrix approximation to the Jacobian.
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory failure).

6. *Problem solution*
7. ARKBANDPRE optional outputs

Optional outputs specific to the SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG solver are listed in *Table: Optional ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG outputs*. To obtain the optional outputs associated with the ARKBANDPRE module, the user should call the `FARKBPOPT()`, as specified below:

subroutine FARKBPOPT(*LENRWBP*, *LENIWBP*, *NFEBP*)

Interfaces with the ARKBANDPRE optional output functions.

Arguments:

- *LENRWBP* (`long int`, output) – length of real preconditioner work space (from `ARKBandPrecGetWorkSpace()`).
- *LENIWBP* (`long int`, output) – length of integer preconditioner work space, in integer words (from `ARKBandPrecGetWorkSpace()`).
- *NFEBP* (`long int`, output) – number of $f_I(t, y)$ evaluations (from `ARKBandPrecGetNumRhsEvals()`)

8. *Additional solution output*
9. *Problem reinitialization*
10. *Memory deallocation*

(The memory allocated for the FARKBP module is deallocated automatically by `FARKFREE()`)

Usage of the FARKBBD interface to ARKBBDPRE

The FARKBBD interface module is a package of C functions which, as part of the FARKODE interface module, support the use of the ARKode solver with the parallel NVECTOR_PARALLEL module, and the combination of the ARKBBDPRE preconditioner module (see the section *A parallel band-block-diagonal preconditioner module*) with any of the Krylov iterative linear solvers.

The user-callable functions in this package, with the corresponding ARKode and ARKBBDPRE functions, are as follows:

- `FARKBBDINIT()` interfaces to `ARKBBDPrecInit()`.

- `FARKBBDREINIT()` interfaces to `ARKBBDPrecReInit()`.
- `FARKBBDOPT()` interfaces to the ARKBBDPRE optional output functions.

In addition to the functions required for general FARKODE usage, the user-supplied functions required by this package are listed in the table below, each with the corresponding interface function which calls it (and its type within ARKBBDPRE or ARKode).

Table: FARKBBD function mapping

FARKBBD routine (FORTRAN, user-supplied)	**ARKode routine (C, interface)**	**ARKode interface function type**
`FARKJTIMES()`	FARKJtimes	`ARKSpilsJacTimesVecFn()`
`FARKGLOCFN()`	FARKgloc	`ARKLocalFn()`
`FARKCOMMFN()`	FARKcfn	`ARKCommFn()`

As with the rest of the FARKODE routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in the section *FARKODE routines*, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `farkbbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in the section *Usage of the FARKODE interface module* are *italicized*.

1. *Right-hand side specification*
2. *NVECTOR module initialization*
3. *Problem specification*
4. *Set optional inputs*
5. Linear solver specification

   First, specify one of the ARKSPILS iterative linear solvers, by calling one of `FARKSPGMR()`, `FARKSPBCG()`, `FARKSPTFQMR()`, `FARKSPFGMR()` or `FARKPCG()`.

   Optionally, to specify that SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG should use the supplied `FARKJTIMES()` routine, the user should call `FARKSPILSSETJAC()` with FLAG $\neq 0$, as described in the section *[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR/SPFGMR/PCG*.

   Then, to initialize the ARKBBDPRE preconditioner, call the function `FARKBBDINIT()`, as described below:

   subroutine FARKBBDINIT(*NLOCAL*, *MUDQ*, *MLDQ*, *MU*, *ML*, *DQRELY*, *IER*)

   Interfaces with the `ARKBBDPrecInit()` routine to initialize the ARKBBDPRE preconditioning module.

   Arguments:

   - *NLOCAL* (`long int`, input) – local vector size on this process.
   - *MUDQ* (`long int`, input) – upper half-bandwidth to be used in the computation of the local Jacobian blocks by difference quotients. These may be smaller than the true half-bandwidths of the Jacobian of the local block of g, when smaller values may provide greater efficiency.
   - *MLDQ* (`long int`, input) – lower half-bandwidth to be used in the computation of the local Jacobian blocks by difference quotients.
   - *MU* (`long int`, input) – upper half-bandwidth of the band matrix that is retained as an approximation of the local Jacobian block (may be smaller than *MUDQ*).
   - *ML* (`long int`, input) – lower half-bandwidth of the band matrix that is retained as an approximation of the local Jacobian block (may be smaller than *MLDQ*).

- *DQRELY* (`realtype`, input) – relative increment factor in y for difference quotients (0.0 indicates to use the default).
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory failure).

6. *Problem solution*
7. ARKBBDPRE optional outputs

   Optional outputs specific to the SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG solver are listed in *Table: Optional ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR and ARKPCG outputs*. To obtain the optional outputs associated with the ARKBBDPRE module, the user should call the `FARKBBDOPT()`, as specified below:

   subroutine `FARKBBDOPT` (*LENRWBBD*, *LENIWBBD*, *NGEBBD*)
   Interfaces with the ARKBBDPRE optional output functions.

   Arguments:

   - *LENRWBP* (`long int`, output) – length of real preconditioner work space on this process (from `ARKBBDPrecGetWorkSpace()`).
   - *LENIWBP* (`long int`, output) – length of integer preconditioner work space on this process (from `ARKBBDPrecGetWorkSpace()`).
   - *NGEBBD* (`long int`, output) – number of $g(t, y)$ evaluations (from `ARKBBDPrecGetNumGfnEvals()`) so far.

8. *Additional solution output*
9. Problem reinitialization

   If a sequence of problems of the same size is being solved using the same linear solver (SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG) in combination with the ARKBBDPRE preconditioner, then the ARKode package can be re-initialized for the second and subsequent problems by calling `FARKREINIT()`, following which a call to `FARKBBDREINIT()` may or may not be needed. If the input arguments are the same, no `FARKBBDREINIT()` call is needed.

   If there is a change in input arguments other than *MU* or *ML*, then the user program should call `FARKBBDREINIT()` as specified below:

   subroutine `FARKBBDREINIT` (*NLOCAL*, *MUDQ*, *MLDQ*, *DQRELY*, *IER*)
   Interfaces with the `ARKBBDPrecReInit()` function to reinitialize the ARKBBDPRE module.

   Arguments: The arguments of the same names have the same meanings as in `FARKBBDINIT()`.

   However, if the value of *MU* or *ML* is being changed, then a call to `FARKBBDINIT()` must be made instead.

   Finally, if there is a change in any of the linear solver inputs, then a call to `FARKSPGMR()`, `FARKSPBCG()`, `FARKSPTFQMR()`, `FARKSPFGMR()` or `FARKPCG()` must also be made; in this case the linear solver memory is reallocated.

10. Problem resizing

    If a sequence of problems of different sizes (but with similar dyanamical time scales) is being solved using the same linear solver (SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG) in combination with the ARKBBDPRE preconditioner, then the ARKode package can be re-initialized for the second and subsequent problems by calling `FARKRESIZE()`, following which a call to `FARKBBDINIT()` is required to delete and re-allocate the preconditioner memory of the correct size.

    subroutine `FARKBBDREINIT` (*NLOCAL*, *MUDQ*, *MLDQ*, *DQRELY*, *IER*)
    Interfaces with the `ARKBBDPrecReInit()` function to reinitialize the ARKBBDPRE module.

    Arguments: The arguments of the same names have the same meanings as in `FARKBBDINIT()`.

However, if the value of MU or ML is being changed, then a call to `FARKBBDINIT()` must be made instead.

Finally, if there is a change in any of the linear solver inputs, then a call to `FARKSPGMR()`, `FARKSPBCG()`, `FARKSPTFQMR()`, `FARKSPFGMR()` or `FARKPCG()` must also be made; in this case the linear solver memory is reallocated.

11. *Memory deallocation*

    (The memory allocated for the FARKBBD module is deallocated automatically by `FARKFREE()`).

12. User-supplied routines

    The following two routines must be supplied for use with the ARKBBDPRE module:

subroutine FARKGLOCFN (*NLOC*, *T*, *YLOC*, *GLOC*, *IPAR*, *RPAR*, *IER*)

User-supplied routine (of type `ARKLocalFn()`) that computes a processor-local approximation $g(t, y)$ to the right-hand side function $f_I(t, y)$.

Arguments:

- *NLOC* (`long int`, input) – local problem size.
- *T* (`realtype`, input) – current value of the independent variable.
- *YLOC* (`realtype`, input) – array containing local dependent state variables.
- *GLOC* (`realtype`, output) – array containing local dependent state derivatives.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

subroutine FARKCOMMFN (*NLOC*, *T*, *YLOC*, *IPAR*, *RPAR*, *IER*)

User-supplied routine (of type `ARKCommFn()`) that performs all interprocess communication necessary for the executation of the `FARKGLOCFN()` function above, using the input vector *YLOC*.

Arguments:

- *NLOC* (`long int`, input) – local problem size.
- *T* (`realtype`, input) – current value of the independent variable.
- *YLOC* (`realtype`, input) – array containing local dependent state variables.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: This subroutine must be supplied even if it is not needed, and must return *IER = 0*.

CHAPTER

SIX

VECTOR DATA STRUCTURES

The SUNDIALS library comes packaged with two NVECTOR implementations, one designed for serial simulations and the second for distributed-memory parallel simulations. Both 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, IDA, 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 and banded linear system solvers, since they rely on particular data storage and access patterns in the NVECTORS used.

Details on the generic NVECTOR module are below. However, to jump to specific descriptions of the serial and parallel vector modules provided by SUNDIALS, or ARKode's requirements for routines comprising a user-supplied NVECTOR module, the following links are provided:

6.1 The NVECTOR_SERIAL Module

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of a `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of data.

```
struct _N_VectorContent_Serial {
   long int length;
   booleantype own_data;
   realtype *data;
};
```

The following five macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix `_S` in the names denotes serial version.

NV_CONTENT_S(v)

This macro gives access to the contents of the serial vector `N_Vector` *v*.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` *content* structure.

Implementation:

```
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

NV_OWN_DATA_S(v)

Access the *own_data* component of the serial `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
```

NV_DATA_S (v)

The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the *data* for the `N_Vector` `v`.

Similarly, the assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
```

NV_LENGTH_S (v)

Access the *length* component of the serial `N_Vector` *v*.

The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the *length* of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the *length* of `v` to be `len_v`.

Implementation:

```
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

NV_Ith_S (v, i)

This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`.

The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to $n-1$ for a vector of length n.

Implementation:

```
#define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in the section *Description of the NVECTOR operations*. Their names are obtained from those in that section by appending the suffix `_Serial`.

In addition, the module NVECTOR_SERIAL provides the following additional user-callable routines:

N_Vector **N_VNew_Serial** (long int *vec_length*)

This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

N_Vector **N_VNewEmpty_Serial** (long int *vec_length*)

This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Serial** (long int *vec_length*, realtype* *v_data*)

This function creates and allocates memory for a serial vector with user-provided data array, *v_data*.

N_Vector* **N_VCloneVectorArray_Serial** (int *count*, N_Vector *w*)

This function creates (by cloning) an array of *count* serial vectors.

N_Vector* **N_VCloneEmptyVectorArray_Serial** (int *count*, N_Vector *w*)

This function creates (by cloning) an array of *count* serial vectors, each with an empty (`'NULL`) data array.

void **N_VDestroyVectorArray_Serial** (N_Vector* *vs*, int *count*)

This function frees memory allocated for the array of *count* variables of type `N_Vector` created with `N_VCloneVectorArray_Serial()` or with `N_VCloneEmptyVectorArray_Serial()`.

void **N_VPrint_Serial** (N_Vector *v*)

This function prints the content of a serial vector to `stdout`.

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.
- `N_VNewEmpty_Serial()`, `N_VMake_Serial()`, and `N_VCloneEmptyVectorArray_Serial()` set the field *own_data* to `FALSE`. The functions `N_VDestroy_Serial()` and `N_VDestroyVectorArray_Serial()` will not attempt to free the pointer data for any `N_Vector` with *own_data* set to `FALSE`. 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.

6.2 The NVECTOR_PARALLEL Module

The parallel implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_PARALLEL, defines the *content* field of a `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, an a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
   long int local_length;
   long int global_length;
   booleantype own_data;
   realtype *data;
   MPI_Comm comm;
};
```

The following seven macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix `_P` in the names denotes 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_VectorParallelContent`.

Implementation:

```
#define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
```

NV_OWN_DATA_P(v)

Access the *own_data* component of the parallel `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_P(v)   ( NV_CONTENT_P(v)->own_data )
```

NV_DATA_P(v)

The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the *local_data* for the `N_Vector v`.

The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data` into *data*.

Implementation:

```
#define NV_DATA_P(v)       ( NV_CONTENT_P(v)->data )
```

NV_LOCLENGTH_P(v)

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`.

The call `NV_LOCLENGTH_P(v) = llen_v` sets the *local_length* of `v` to be `llen_v`.

Implementation:

```
#define NV_LOCLENGTH_P(v)  ( NV_CONTENT_P(v)->local_length )
```

NV_GLOBLENGTH_P(v)

The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the *global_length* of the vector `v`.

The call `NV_GLOBLENGTH_P(v) = glen_v` sets the *global_length* of `v` to be `glen_v`.

Implementation:

```
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

NV_COMM_P(v)

This macro provides access to the MPI communicator used by the parallel `N_Vector` *v*.

Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

NV_Ith_P(v, i)

This macro gives access to the individual components of the *local_data* array of an `N_Vector`.

The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`.

The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

Here `i` ranges from 0 to $n - 1$, where n is the *local_length*.

Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in the section *Description of the NVECTOR operations*. Their names are obtained from those that section by appending the suffix `_Parallel`.

In addition, the module NVECTOR_PARALLEL provides the following additional user-callable routines:

N_Vector **N_VNew_Parallel**(MPI_Comm *comm*, long int *local_length*, long int *global_length*)

This function creates and allocates memory for a parallel vector having global length *global_length*, having processor-local length *local_length*, and using the MPI communicator *comm*.

N_Vector **N_VNewEmpty_Parallel**(MPI_Comm *comm*, long int *local_length*, long int *global_length*)

This function creates a new parallel `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Parallel**(MPI_Comm *comm*, long int *local_length*, long int *global_length*, realtype* *v_data*)

This function creates and allocates memory for a parallel vector with user-provided data array.

N_Vector* **N_VCloneVectorArray_Parallel**(int *count*, N_Vector *w*)

This function creates (by cloning) an array of *count* parallel vectors.

N_Vector* **N_VCloneEmptyVectorArray_Parallel**(int *count*, N_Vector *w*)

This function creates (by cloning) an array of *count* parallel vectors, each with an empty (`NULL`) data array.

void **N_VDestroyVectorArray_Parallel** (N_Vector* *vs*, int *count*)
: This function frees memory allocated for the array of *count* variables of type `N_Vector` created with `N_VCloneVectorArray_Parallel()` or with `N_VCloneEmptyVectorArray_Parallel()`.

void **N_VPrint_Parallel** (N_Vector *v*)
: This function prints the content of a parallel vector to `stdout`.

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.
- `N_VNewEmpty_Parallel()`, `N_VMake_Parallel()`, and `N_VCloneEmptyVectorArray_Parallel()` set the field *own_data* to `FALSE`. The routines `N_VDestroy_Parallel()` and `N_VDestroyVectorArray_Parallel()` will not attempt to free the pointer data for any `N_Vector` with *own_data* set to `FALSE`. 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.

6.3 NVECTOR functions required by ARKode

In the table below, we list the vector functions in the `N_Vector` module that are called within the ARKode package. The table also shows, for each function, which ARKode module uses the function: the ARKde column shows function usage within the main integrator module, while the remaining columns show function usage within the ARKode linear solver modules (ARKSPILS stands for any of ARKSPGMR, ARKSPBCG, ARKSPTFQMR, ARKSPFGMR or ARKPCG), the ARKBANDPRE and ARKBBDPRE preconditioner modules, and the FARKODE module.

Routine	**ARKode**	**ARK-DENSE**	**ARK-BAND**	**ARK-SPILS**	**ARKBAND-PRE**	**ARKBBD-PRE**	**FARKODE**
N_VAbs	X						
N_VAddConst	X						
N_VClone	X			X			
N_VCloneEmpty							X
N_VConst	X	X	X	X			
N_VDestroy	X			X			X
N_VDiv	X			X			
N_VDotProd	X			X			
N_VGetArrayPointer		X	X		X	X	X
N_VInv	X						
N_VLinearSum	X	X		X			
N_VMaxNorm	X						
N_VMin	X						X
N_VProd				X			
N_VScale	X	X	X	X	X	X	
N_VSetArrayPointer		X					X
N_VSpace	X						
N_VWrmsNorm	X	X	X	X	X	X	

At this point, we should emphasize that the ARKode user does not need to know anything about the usage of vector functions by the ARKode code modules in order to use ARKode. Instead, this information is provided primarily for users interested in constructing a custom `N_Vector` module. We note that a number of `N_Vector` functions from the section *Description of the NVECTOR Modules* are not listed in the above table. Therefore a user-supplied N_Vector module for ARKode could safely omit these functions from their implementation. Furthermore, since the N_VSpace() function is only informational, while it should be supplied, the return values may be arbitrary.

6.4 Description of the NVECTOR Modules

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type N_Vector) through a set of operations defined by, and specific to, the particular NVECTOR implementation. Users can provide a custom implementation of the NVECTOR module or use one of two provided within SUNDIALS, one serial and the other an MPI parallel implementation.

The generic N_Vector type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type N_Vector is defined as:

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
   void *content;
   struct _generic_N_Vector_Ops *ops;
};
```

Here, the _generic_N_Vector_Op structure is essentially a list of function pointers to the various actual vector operations, and is defined as

```
struct _generic_N_Vector_Ops {
   N_Vector     (*nvclone)(N_Vector);
   N_Vector     (*nvcloneempty)(N_Vector);
   void         (*nvdestroy)(N_Vector);
   void         (*nvspace)(N_Vector, long int *, long int *);
   realtype*    (*nvgetarraypointer)(N_Vector);
   void         (*nvsetarraypointer)(realtype *, N_Vector);
   void         (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
   void         (*nvconst)(realtype, N_Vector);
   void         (*nvprod)(N_Vector, N_Vector, N_Vector);
   void         (*nvdiv)(N_Vector, N_Vector, N_Vector);
   void         (*nvscale)(realtype, N_Vector, N_Vector);
   void         (*nvabs)(N_Vector, N_Vector);
   void         (*nvinv)(N_Vector, N_Vector);
   void         (*nvaddconst)(N_Vector, realtype, N_Vector);
   realtype     (*nvdotprod)(N_Vector, N_Vector);
   realtype     (*nvmaxnorm)(N_Vector);
   realtype     (*nvwrmsnorm)(N_Vector, N_Vector);
   realtype     (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
   realtype     (*nvmin)(N_Vector);
   realtype     (*nvwl2norm)(N_Vector, N_Vector);
   realtype     (*nvl1norm)(N_Vector);
   void         (*nvcompare)(realtype, N_Vector, N_Vector);
   booleantype  (*nvinvtest)(N_Vector, N_Vector);
   booleantype  (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
   realtype     (*nvminquotient)(N_Vector, N_Vector);
};
```

The generic NVECTOR module defines and implements the vector operations acting on a N_Vector. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the N_Vector structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```
void N_VScale(realtype c, N_Vector x, N_Vector z) {
   z->ops->nvscale(c, x, z);
}
```

The subsection *Description of the NVECTOR operations* contains a complete list of all vector operations defined by the generic NVECTOR module. Finally, we note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneEmptyVectorArray`. Both functions create (by cloning) an array of *count* variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are:

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneEmptyVectorArray(int count, N_Vector w);
```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

Similarly, an array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

```
void N_VDestroyVectorArray(N_Vector *vs, int count);
```

and whose definition is based on the implementation-specific `N_VDestroy` operation.

In particular, any implementation of the NVECTOR module **must**:

- Specify the *content* field of the `N_Vector`.
- Define and implement the necessary vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code. We further note that not all of the defined operations are required for each solver in SUNDIALS. The list of required operations for use with ARKode is given in the section *NVECTOR functions required by ARKode*.
- Define and implement user-callable constructor and destructor routines to create and free a `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the *content* for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the content field of the newly defined `N_Vector`.

6.5 Description of the NVECTOR operations

For each of the `N_vector` operations, we give the name, usage of the function, and a description of its mathematical operations below.

N_Vector **`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 array.

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*, long int* *lrw*, long int* *liw*)
Returns storage requirements for the `N_Vector` *v*: *lrw* contains the number of `realtype` words and *liw* contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.

Usage:

```
N_VSpace(nvSpec, &lrw, &liw);
```

realtype* **N_VGetArrayPointer** (N_Vector *v*)
Returns a pointer to a `realtype` array from the `N_Vector` *v*. Note that this assumes that the internal data in the `N_Vector` is a contiguous array of `realtype`. This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.

Usage:

```
vdata = NVGetArrayPointer(v);
```

void **N_VSetArrayPointer** (realtype* *vdata*, N_Vector *v*)
Replaces the data array pointer in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in the `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module.

Usage:

```
NVSetArrayPointer(vdata,v);
```

void **N_VLinearSum** (realtype *a*, N_Vector *x*, realtype *b*, N_Vector *y*, N_Vector *z*)
Performs the operation $z = ax + by$, where *a* and *b* are scalars and *x* and *y* are of type `N_Vector`:

$$z_i = ax_i + by_i, \quad i = 1, \ldots, n.$$

Usage:

```
N_VLinearSum(a, x, b, y, z);
```

void **N_VConst** (realtype *c*, N_Vector *z*)
Sets all components of the `N_Vector` *z* to the scalar value *c*:

$$z_i = c, \quad i = 1, \ldots, n.$$

Usage:

```
N_VConst(c, z);
```

void **N_VProd** (N_Vector *x*, N_Vector *y*, N_Vector *z*)

Sets the `N_Vector` *z* to be the component-wise product of the `N_Vector` inputs *x* and *y*:

$$z_i = x_i y_i, \quad i = 1, \ldots, n.$$

Usage:

```
N_VProd(x, y, z);
```

void **N_VDiv** (N_Vector *x*, N_Vector *y*, N_Vector *z*)

Sets the `N_Vector` *z* to be the component-wise ratio of the `N_Vector` inputs *x* and *y*:

$$z_i = \frac{x_i}{y_i}, \quad i = 1, \ldots, n.$$

The y_i may not be tested for 0 values. It should only be called with a *y* that is guaranteed to have all nonzero components.

Usage:

```
N_VDiv(x, y, z);
```

void **N_VScale** (realtype *c*, N_Vector *x*, N_Vector *z*)

Scales the `N_Vector` *x* by the scalar *c* and returns the result in *z*:

$$z_i = c x_i, \quad i = 1, \ldots, n.$$

Usage:

```
N_VScale(c, x, z);
```

void **N_VAbs** (N_Vector *x*, N_Vector *z*)

Sets the components of the `N_Vector` *z* to be the absolute values of the components of the `N_Vector` *x*:

$$y_i = |x_i|, \quad i = 1, \ldots, n.$$

Usage:

```
N_VAbs(x, z);
```

void **N_VInv** (N_Vector *x*, N_Vector *z*)

Sets the components of the `N_Vector` *z* to be the inverses of the components of the `N_Vector` *x*:

$$z_i = 1.0/x_i, \quad i = 1, \ldots, n.$$

This routine may not check for division by 0. It should be called only with an *x* which is guaranteed to have all nonzero components.

Usage:

```
N_VInv(x, z);
```

void **`N_VAddConst`** (N_Vector *x*, realtype *b*, N_Vector *z*)

Adds the scalar *b* to all components of *x* and returns the result in the `N_Vector` *z*:

$$z_i = x_i + b, \quad i = 1, \ldots, n.$$

Usage:

```
N_VAddConst(x, b, z);
```

realtype **`N_VDotProd`** (N_Vector *x*, N_Vector *z*)

Returns the value of the dot-product of the `N_Vectors` *x* and *y*:

$$d = \sum_{i=1}^{n} x_i y_i.$$

Usage:

```
d = N_VDotProd(x, y);
```

realtype **`N_VMaxNorm`** (N_Vector *x*)

Returns the value of the l_∞ norm of the `N_Vector` *x*:

$$m = \max_{1 \le i \le n} |x_i|.$$

Usage:

```
m = N_VMaxNorm(x);
```

realtype **`N_VWrmsNorm`** (N_Vector *x*, N_Vector *w*)

Returns the weighted root-mean-square norm of the `N_Vector` *x* with (positive) weight vector *w*:

$$m = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i w_i)^2 \right)^{1/2}.$$

Usage:

```
m = N_VWrmsNorm(x, w);
```

realtype **`N_VWrmsNormMask`** (N_Vector *x*, N_Vector *w*, N_Vector *id*)

Returns the weighted root mean square norm of the `N_Vector` *x* with (positive) weight vector *w* built using only the elements of *x* corresponding to nonzero elements of the `N_Vector` *id*:

$$m = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i w_i \text{sign}(id_i))^2 \right)^{1/2}.$$

```
m = N_VWrmsNormMask(x, w, id);
```

realtype **N_VMin** (N_Vector *x*)

Returns the smallest element of the `N_Vector` *x*:

$$m = \min_{1\le i\le n} x_i.$$

Usage:

```
m = N_VMin(x);
```

realtype **N_VWl2Norm** (N_Vector *x*, N_Vector *w*)

Returns the weighted Euclidean l_2 norm of the `N_Vector` *x* with weight vector *w*:

$$m = \left(\sum_{i=1}^{n} (x_i w_i)^2\right)^{1/2}.$$

Usage:

```
m = N_VWL2Norm(x, w);
```

realtype **N_VL1Norm** (N_Vector *x*)

Returns the l_1 norm of the `N_Vector` *x*:

$$m = \sum_{i=1}^{n} |x_i|.$$

Usage:

```
m = N_VL1Norm(x);
```

void **N_VCompare** (realtype *c*, N_Vector *x*, N_Vector *z*)

Compares the components of the `N_Vector` *x* to the scalar *c* and returns an `N_Vector` *z* such that for all $1 \le i \le n$,

$$z_i = \begin{cases} 1.0 & \text{if } |x_i| \ge c, \\ 0.0 & \text{otherwise} \end{cases}.$$

Usage:

```
N_VCompare(c, x, z);
```

booleantype **N_VInvTest** (N_Vector *x*, N_Vector *z*)

Sets the components of the `N_Vector` *z* to be the inverses of the components of the `N_Vector` *x*, with prior testing for zero values:

$$z_i = 1.0/x_i, \quad i = 1, \dots, n.$$

This routine returns `TRUE` if all components of *x* are nonzero (successful inversion) and returns `FALSE` otherwise.

Usage:

```
t = N_VInvTest(x, z);
```

booleantype **N_VConstrMask** (N_Vector *c*, N_Vector *x*, N_Vector *m*)

Performs the following constraint tests based on the values in c_i:

$$\begin{aligned} x_i > 0 \text{ if } c_i = 2, \\ x_i \geq 0 \text{ if } c_i = 1, \\ x_i < 0 \text{ if } c_i = -2, \\ x_i \leq 0 \text{ if } c_i = -1. \end{aligned}$$

There is no constraint on x_i if $c_i = 0$. This routine returns `FALSE` if any element failed the constraint test, `TRUE` if all passed. It also sets a mask vector *m*, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Usage:

```
t = N_VConstrMask(c, x, m);
```

realtype **N_VMinQuotient** (N_Vector *num*, N_Vector *denom*)

This routine returns the minimum of the quotients obtained by termwise dividing the elements of *n* by the elements in *d*:

$$\min_{i=1,\ldots,n} \frac{\text{num}_i}{\text{denom}_i}.$$

A zero element in *denom* will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

Usage:

```
minq = N_VMinQuotient(num, denom);
```

CHAPTER

SEVEN

LINEAR SOLVERS IN ARKODE

In this section, we describe the generic linear solver code modules from SUNDIALS that are included in ARKode. While these may be used in conjunction with ARKode, they may also be used separately as generic packages in themselves. These generic linear solver modules are organized in two families of solvers, the DLS family, which includes direct linear solvers appropriate for sequential computations; and the SPILS family, which includes scaled preconditioned iterative (Krylov) linear solvers. The solvers in each family share common data structures and functions.

The *DLS* family contains the following two generic linear solvers:

- The DENSE package, a linear solver for dense matrices either specified through a matrix type (defined below) or as simple arrays.
- The BAND package, a linear solver for banded matrices either specified through a matrix type (defined below) or as simple arrays.

We further note that this family also includes the BLAS/LAPACK linear solvers (dense and band) available to the SUNDIALS solvers, but these are not discussed here.

The *SPILS* family contains the following generic linear solvers:

- The SPGMR package, a solver for the scaled preconditioned GMRES method.
- The SPBCG package, a solver for the scaled preconditioned Bi-CGStab method.
- The SPTFQMR package, a solver for the scaled preconditioned TFQMR method.
- The SPFGMR package, a solver for the scaled preconditioned Flexible GMRES method.
- The PCG package, a solver for the preconditioned conjugate gradient method.

For reasons related to installation, the names of the files involved in these generic solvers begin with the prefix SUNDIALS. But despite this, each of the solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

For the sake of space, the functions for the DENSE and BAND modules that work with a matrix type and the functions in the SPGMR, SPBCG, SPTFQMR, SPFGMR and PCG modules are only summarized briefly, since they are less likely to be of direct use in connection with a SUNDIALS solver. However, the functions for dense matrices treated as simple arrays are fully described, because we anticipate that they will be useful in the implementation of preconditioners used with the combination of one of the SUNDIALS solvers and one of the SPILS linear solvers.

Lastly, it is possible to supply customized linear solvers to ARKode, in that the ARKode solvers only require the existence of a minimal set of generic routines. Through attaching user-supplied routines for these function pointers, it is possible to use arbitrary approaches for solution to the implicit linear systems arising during an ARKode solve.

Specifics of these built-in linear solver packages, as well as the generic linear solver interface, are provided in the following sub-sections:

7.1 The DLS modules: DENSE and BAND

The files comprising the DENSE generic linear solver, and their locations in the SUNDIALS `srcdir`, are as follows:

- header files (located in `srcdir/include/sundials`):

  `sundials_direct.h`, `sundials_dense.h`, `sundials_types.h`, `sundials_math.h`, `sundials_config.h`

- source files (located in `srcdir/src/sundials`):

  `sundials_direct.c`, `sundials_dense.c`, `sundials_math.c`

The files comprising the BAND generic linear solver are as follows:

- header files (located in `srcdir/include/sundials`):

  `sundials_direct.h`, `sundials_band.h`, `sundials_types.h`, `sundials_math.h`, `sundials_config.h`

- source files (located in `srcdir/src/sundials`):

  `sundials_direct.c`, `sundials_band.c`, `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the DENSE and BAND packages by themselves (see the section *ARKode Installation Procedure* for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:

  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```

- (optional) use of generic math functions:

  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MIN`, `MAX`, and `ABS` macros and `RAbs` and `RSqrt` functions.

The files listed above for either module can be extracted from the SUNDIALS `srcdir` and compiled by themselves into a separate library or into a larger user code.

7.1.1 DlsMat

The type `DlsMat`, defined in `sundials_direct.h` is a pointer to a structure defining a generic matrix, and is used with all linear solvers in the DLS family:

`DlsMat`

```
typedef struct _DlsMat {
  int type;
  long int M;
  long int N;
  long int ldim;
  long int mu;
  long int ml;
  long int s_mu;
```

```
  realtype *data;
  long int ldata;
  realtype **cols;
} *DlsMat;
```

For the DENSE module, the relevant fields of this structure are as follows. Note that a dense matrix of type `DlsMat` need not be square.

type – `SUNDIALS_DENSE` (=1)

M – number of rows

N – number of columns

ldim – leading dimension ($\geq M$)

data – pointer to a contiguous block of `realtype` variables

ldata – length of the data array ($= ldim * N$). The `(i,j)` element of a dense matrix `A` of type `DlsMat` (with $0 \leq i < M$ and $0 \leq j < N$) is given by the expression `(A->data)[0][j*M+i]`

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 matrix `A` of type `DlsMat` (with $0 \leq i < M$ and $0 \leq j < N$) is given by the expression `(A->cols)[j][i]`

For the BAND module, the relevant fields of this structure are as follows (see Figure *DLS Diagram* for a diagram of the underlying data representation in a banded matrix of type `DlsMat`). Note that only square band matrices are allowed.

type – `SUNDIALS_BAND` (=2)

M – number of rows

N – number of columns ($N = M$)

mu – upper half-bandwidth, $0 \leq mu < min(M, N)$

ml – lower half-bandwidth, $0 \leq ml < min(M, N)$

s_mu – storage upper bandwidth, $mu \leq s\_mu < N$. The LU decomposition routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as $min(N - 1, mu + ml)$ because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for A.

ldim – leading dimension ($ldim \geq s\_mu$)

data – pointer to a contiguous block of `realtype` variables. The elements of a banded matrix of type `DlsMat` 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 band of A.

ldata – length of the `data` array ($= ldim * (s\_mu + ml + 1)$)

cols – array of pointers. `cols[j]` is a pointer to the uppermost element within the band in the `j`-th column. This pointer may be treated as an array indexed from `s_mu-mu` (to access the uppermost element within the band in the `j`-th column) to `s_mu+ml` (to access the lowest element within the band in the `j`-th column). Indices from 0 to `s_mu-mu-1` give access to extra storage elements required by the LU decomposition function. Finally, `cols[j][i-j+s_mu]` is the `(i,j)`-th element, $j - mu \leq i \leq j + ml$.

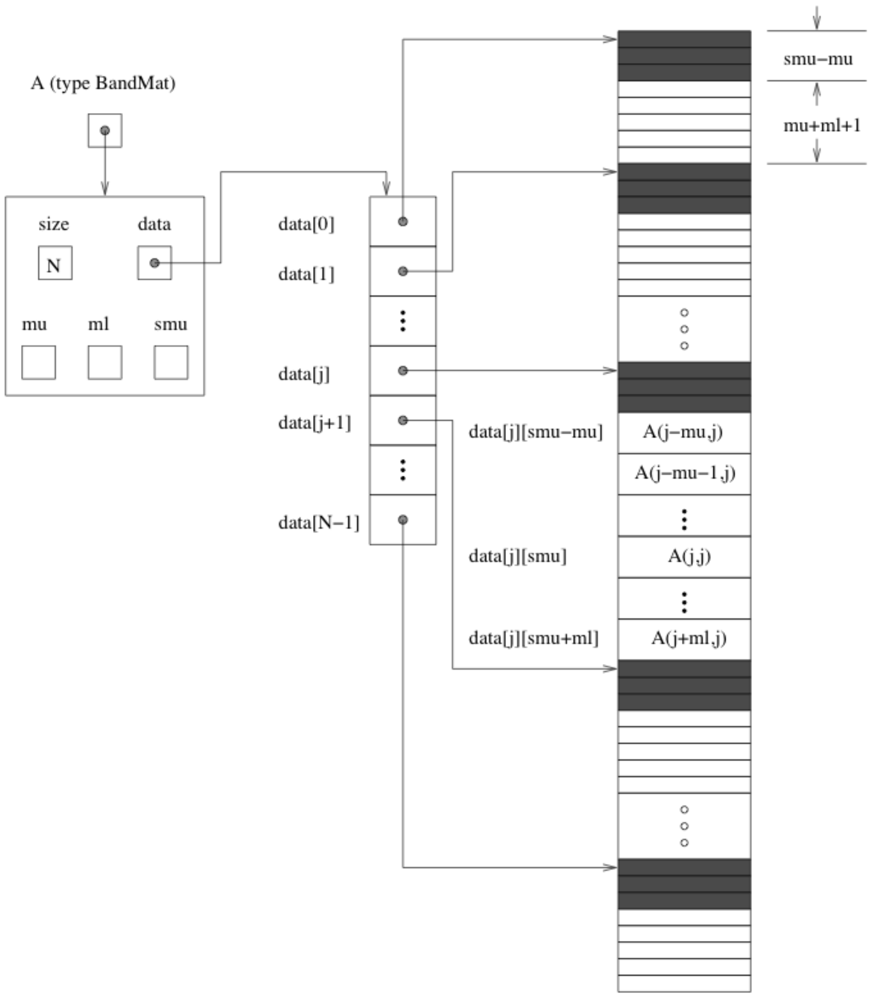


Figure 7.1: DLS Diagram: Storage for a banded matrix of type DlsMat. Here A is an $N \times N$ band matrix of type DlsMat with upper and lower half-bandwidths mu and ml, respectively. The rows and columns of A are numbered from 0 to $N-1$ and the (i, j)-th element of A is denoted A(i, j). The greyed out areas of the underlying component storage are used by the BandGBTRF and BandGBTRS routines.

7.1.2 Accessor macros for the DLS modules

The macros below allow a user to efficiently access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer to the j-th column of elements can be obtained via the `DENSE_COL` or `BAND_COL` macros. Users should use these macros whenever possible.

The following two macros are defined by the DENSE module to provide access to data in the `DlsMat` type:

DENSE_ELEM

Usage: `DENSE_ELEM(A,i,j) = a_ij;` or `a_ij = DENSE_ELEM(A,i,j);`

This macro references the (i, j)-th element of the $M \times N$ `DlsMat` A, $0 \le i < M$, $0 \le j < N$.

DENSE_COL

Usage: `col_j = DENSE_COL(A,j);`

This macro references the j-th column of the $M \times N$ `DlsMat` A, $0 \le j < N$. The type of the expression `DENSE_COL(A,j)` is `realtype *`. After the assignment in the usage above, `col_j` may be treated as an array indexed from 0 to $M - 1$. The (i, j)-th element of A is referenced by `col_j[i]`.

The following three macros are defined by the BAND module to provide access to data in the `DlsMat` type:

BAND_ELEM

Usage: `BAND_ELEM(A,i,j) = a_ij;` or `a_ij = BAND_ELEM(A,i,j);`

This macro references the (i, j)-th element of the $N \times N$ band matrix A, where $0 \le i, j \le N - 1$. The location (i, j) should further satisfy $j-$ `(A->mu)` $\le i \le j+$ `(A->ml)`.

BAND_COL

Usage: `col_j = BAND_COL(A,j);`

This macro references the diagonal element of the j-th column of the $N \times N$ band matrix A, $0 \le j \le N - 1$. The type of the expression `BAND_COL(A,j)` is `realtype *`. The pointer returned by the call `BAND_COL(A,j)` can be treated as an array which is indexed from `-(A->mu)` to `(A->ml)`.

BAND_COL_ELEM

Usage: `BAND_COL_ELEM(col_j,i,j) = a_ij;` or `a_ij = BAND_COL_ELEM(col_j,i,j);`

This macro references the (i, j)-th entry of the band matrix A when used in conjunction with `BAND_COL` to reference the j-th column through `col_j`. The index (i, j) should satisfy $j-$ `(A->mu)` $\le i \le j+$ `(A->ml)`.

7.1.3 Functions in the DENSE module

The DENSE module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on dense matrices of type `DlsMat`. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for DlsMat dense matrices are available in the DENSE package. For full details, see the header files `sundials_direct.h` and `sundials_dense.h`.

DlsMat **`NewDenseMat`** (long int M, long int N)

Allocates a `DlsMat` dense matrix.

void **`DestroyMat`** (DlsMat A)

Frees memory for a `DlsMat` matrix

void **`PrintMat`** (DlsMat A)

Prints a `DlsMat` matrix to standard output.

long int* **`NewLintArray`** (long int N)

Allocates an array of `long int` integers for use as pivots with `DenseGETRF()` and `DenseGETRS()`.

int* **`NewIntArray`**(int *N*)
: Allocates an array of `int` integers for use as pivots with the LAPACK dense solvers.

realtype* **`NewRealArray`**(long int *N*)
: Allocates an array of type `realtype` for use as right-hand side with `DenseGETRS()`.

void **`DestroyArray`**(void* *p*)
: Frees memory for an array.

void **`SetToZero`**(DlsMat *A*)
: Loads a matrix with zeros.

void **`AddIdentity`**(DlsMat *A*)
: Increments a square matrix by the identity matrix.

void **`DenseCopy`**(DlsMat *A*, DlsMat *B*)
: Copies one dense matrix to another.

void **`DenseScale`**(realtype *c*, DlsMat *A*)
: Scales a dense matrix by a scalar.

long int **`DenseGETRF`**(DlsMat *A*, long int* *p*)
: LU factorization with partial pivoting of a dense matrix.

long int **`denseGETRF`**(realtype** *a*, long int *m*, long int *n*, long int* *p*)
: Solves $Ax = b$ using LU factorization (for square matrices A), using the factorization resulting from `DenseGETRF()`.

long int **`DensePOTRF`**(DlsMat *A*)
: Cholesky factorization of a real symmetric positive definite dense matrix.

void **`DensePOTRS`**(DlsMat *A*, realtype* *b*)
: Solves $Ax = b$ using the Cholesky factorization of A resulting from a call to `DensePOTRF()`.

int **`DenseGEQRF`**(DlsMat *A*, realtype* *beta*, realtype* *wrk*)
: QR factorization of an $m \times n$ dense matrix, with $m \geq n$.

int **`DenseORMQR`**(DlsMat *A*, realtype* *beta*, realtype* *vn*, realtype* *vm*, realtype* *wrk*)
: Computes the product $w = Qv$, with Q calculated using `DenseGEQRF()`.

The following functions for small dense matrices are available in the DENSE package. These functions primarily replicate those defined above for `DlsMat` dense matrices, but act on the individual data arrays outside of the `DlsMat` structure:

realtype** **`newDenseMat`**(long int *m*, long int *n*)
: Allocates storage for an $m \times n$ dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then the function returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = newDenseMat(m,n)`, then `a[j][i]` references the row `i`, column `j` element of the matrix `a`, $0 \leq i < m$, $0 \leq j < n$, and `a[j]` is a pointer to the first element in the j-th column of `a`. The location `a[0]` contains a pointer to $m \times n$ contiguous locations which contain the elements of `a`.

void **`destroyMat`**(realtype** *a*)
: Frees the dense matrix *a* allocated by `newDenseMat()`.

long int* **`newLintArray`**(long int *n*)
: Allocates an array of *n* integers of `long int` type. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

int* **`newIntArray`**(int *n*)
: Allocates an array of *n* integers of type `int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

realtype* **newRealArray** (long int *m*)
Allocates an array of *n* `realtype` values. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

void **destroyArray** (void* *v*)
Frees the array *v* allocated by `newLintArray()`, `newIntArray()`, or `newRealArray()`.

void **denseCopy** (realtype** *a*, realtype** *b*, long int *m*, long int *n*)
Copies the $m \times n$ dense matrix *a* into the $m \times n$ dense matrix *b*.

void **denseScale** (realtype *c*, realtype** *a*, long int *m*, long int *n*)
Scales every element in the $m \times n$ dense matrix *a* by the scalar *c*.

void **denseAddIdentity** (realtype** *a*, long int *n*)
Increments the square $n \times n$ dense matrix *a* by the identity matrix I_n.

long int **denseGETRF** (realtype** *a*, long int *m*, long int *n*, long int* *p*)
Factors the $m \times n$ dense matrix *a*, using Gaussian elimination with row pivoting. It overwrites the elements of *a* with its LU factors and keeps track of the pivot rows chosen in the pivot array *p*.

A successful LU factorization leaves the matrix *a* and the pivot array *p* with the following information:

1. `p[k]` contains the row number of the pivot element chosen at the beginning of elimination step $k, k = 0, 1, \ldots, n-1$.
2. If the unique LU factorization of *a* is given by $Pa = LU$, where P is a permutation matrix, L is a $m \times n$ lower trapezoidal matrix with all diagonal elements equal to 1, and U is a $n \times n$ upper triangular matrix, then the upper triangular part of *a* (including its diagonal) contains U and the strictly lower trapezoidal part of *a* contains the multipliers, $I - L$. If *a* is square, L is a unit lower triangular matrix.

   `denseGETRF()` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix a does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

void **denseGETRS** (realtype** *a*, long int *n*, long int* *p*, realtype* *b*)
Solves the $n \times n$ linear system $ax = b$. It assumes that *a* (of size $n \times n$) has been LU-factored and the pivot array *p* has been set by a successful call to `denseGETRF()`. The solution *x* is written into the *b* array.

long int **densePOTRF** (realtype** *a*, long int *m*)
Calculates the Cholesky decomposition of the $m \times m$ dense matrix *a*, assumed to be symmetric positive definite. Only the lower triangle of *a* is accessed and overwritten with the Cholesky factor.

void **densePOTRS** (realtype** *a*, long int *m*, realtype* *b*)
Solves the $m \times m$ linear system $ax = b$. It assumes that the Cholesky factorization of *a* has been calculated in the lower triangular part of *a* by a successful call to `densePOTRF(m)()`.

int **denseGEQRF** (realtype** *a*, long int *m*, long int *n*, realtype* *beta*, realtype* *v*)
Calculates the QR decomposition of the $m \times n$ matrix *a* ($m \geq n$) using Householder reflections. On exit, the elements on and above the diagonal of *a* contain the $n \times n$ upper triangular matrix R; the elements below the diagonal, with the array *beta*, represent the orthogonal matrix Q as a product of elementary reflectors. The real array *wrk*, of length *m*, must be provided as temporary workspace.

int **denseORMQR** (realtype** *a*, long int *m*, long int *n*, realtype* *beta*, realtype* *v*, realtype* *w*, realtype* *wrk*)
Calculates the product $w = Qv$ for a given vector *v* of length *n*, where the orthogonal matrix Q is encoded in the $m \times n$ matrix *a* and the vector *beta* of length *n*, after a successful call to `denseGEQRF()`. The real array *wrk*, of length *m*, must be provided as temporary workspace.

7.1.4 Functions in the BAND module

The BAND module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on band matrices of type `DlsMat`. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for `DlsMat` banded matrices are available in the BAND package. For full details, see the header files `sundials_direct.h` and `sundials_band.h`. A number of these are shared with routines from the DENSE package, but are listed again here for completeness.

DlsMat **`NewBandMat`** (long int *N*, long int *mu*, long int *ml*, long int *smu*)
: Allocates a `DlsMat` band matrix

void **`DestroyMat`** (DlsMat *A*)
: Frees memory for a `DlsMat` matrix

void **`PrintMat`** (DlsMat *A*)
: Prints a `DlsMat` matrix to standard output.

long int* **`NewLintArray`** (long int *N*)
: Allocates an array of `long int` integers for use as pivots with `BandGBRF()` and `BandGBRS()`.

int* **`NewIntArray`** (int *N*)
: Allocates an array of `int` integers for use as pivots with the LAPACK band solvers.

realtype* **`NewRealArray`** (long int *N*)
: Allocates an array of type `realtype` for use as right-hand side with `BandGBRS()`.

void **`DestroyArray`** (void* *p*)
: Frees memory for an array.

void **`SetToZero`** (DlsMat *A*)
: Loads a matrix with zeros.

void **`AddIdentity`** (DlsMat *A*)
: Increments a square matrix by the identity matrix.

void **`BandCopy`** (DlsMat *A*, DlsMat *B*, long int *copymu*, long int *copyml*)
: Copies one band matrix to another.

void **`BandScale`** (realtype *c*, DlsMat *A*)
: Scales a band matrix by a scalar.

long int **`BandGBTRF`** (DlsMat *A*, long int* *p*)
: LU factorization with partial pivoting.

void **`BandGBTRS`** (DlsMat *A*, long int* *p*, realtype* *b*)
: Solves $Ax = b$ using LU factorization resulting from `BandGBTRF()`.

The following functions for small band matrices are available in the BAND package. These functions primarily replicate those defined above for `DlsMat` banded matrices, but act on the individual data arrays outside of the `DlsMat` structure:

realtype** **`newBandMat`** (long int *n*, long int *smu*, long int *ml*)
: Allocates storage for a $n \times n$ band matrix with lower half-bandwidth *ml*.

void **`destroyMat`** (realtype** *a*)
: Frees the band matrix *a* allocated by `newBandMat()`.

long int* **`newLintArray`** (long int *n*)
: Allocates an array of *n* integers of type `long int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

int* **newIntArray** (int *n*)

Allocates an array of *n* integers of type `int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

realtype* **newRealArray** (long int *m*)

Allocates an array of *n* `realtype` values. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

void **destroyArray** (void* *v*)

Frees the array *v* allocated by `newLintArray()`, `newIntArray()`, or `newRealArray()`.

void **bandCopy** (realtype** *a*, realtype** *b*, long int *n*, long int *a_smu*, long int *b_smu*, long int *copymu*, long int *copyml*)

Copies the $n \times n$ band matrix *a* into the $n \times n$ band matrix *b*.

void **bandScale** (realtype *c*, realtype** *a*, long int *n*, long int *mu*, long int *ml*, long int *smu*)

Scales every element in the $n \times n$ band matrix *a* by *c*.

void **bandAddIdentity** (realtype** *a*, long int *n*, long int *smu*)

Increments the $n \times n$ band matrix *a* by the identity matrix.

long int **bandGBTRF** (realtype** *a*, long int *n*, long int *mu*, long int *ml*, long int *smu*, long int* *p*)

Factors the $n \times n$ band matrix *a*, using Gaussian elimination with row pivoting. It overwrites the elements of *a* with its LU factors and keeps track of the pivot rows chosen in the pivot array *p*.

void **bandGBTRS** (realtype** *a*, long int *n*, long int *smu*, long int *ml*, long int* *p*, realtype* *b*)

Solves the $n \times n$ linear system $ax = b$. It assumes that *a* (of size $n \times n$) has been LU-factored and the pivot array *p* has been set by a successful call to `bandGETRF()`. The solution *x* is written into the *b* array.

7.2 The SPILS modules: SPGMR, SPBCG, SPTFQMR, SPFGMR and PCG

Due to their reliance on only general vector operations (without need to directly access data), the iterative linear solvers in the SPILS family can be used with a relatively complete NVECTOR implementation (see the section *NVECTOR functions required by ARKode* for a complete listing). We note that while both the *NVECTOR_SERIAL* and *NVECTOR_PARALLEL* modules provided with SUNDIALS meet these criteria, these criteria may also be easily met through a user-supplied vector implementation.

In the subsections below, we discuss the iterative linear solvers accessible to ARKode, SPGMR, SPBCG, SPTFQMR, SPFGMR and PCG. Due to the similarities between these modules, we provide a more complete description of only the SPGMR interface, and for the remaining solvers only discuss the salient differences.

7.2.1 The SPGMR module

The SPGMR package, in the files `sundials_spgmr.h` and `sundials_spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in `sundials_iterative.h` and `sundials_iterative.c`, contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPBCG, SPTFQMR, SPFGMR and PCG). For full details, including usage instructions, see the header files `sundials_spgmr.h` and `sundials_iterative.h`.

The files comprising the SPGMR generic linear solver, and their locations in the SUNDIALS `srcdir`, are as follows:

- header files (located in `srcdir/include/sundials`):

  `sundials_spgmr.h`, `sundials_iterative.h`, `sundials_nvector.h`, `sundials_types.h`, `sundials_math.h`, `sundials_config.h`

- source files (located in `srcdir/src/sundials`):

  `sundials_spgmr.c`, `sundials_iterative.c`, `sundials_nvector.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the SPGMR package by itself (see the section *ARKode Installation Procedure* for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following three lines must be present:

  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```

- (optional) use of generic math functions:

  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MAX` and `ABS` macros and `RAbs` and `RSqrt` functions.

The generic NVECTOR files, `sundials_nvector.h` and `sundials_nvector.c` are needed for the definition of the generic `N_Vector` type and functions. The NVECTOR functions used by the SPGMR module are: `N_VDotProd()`, `N_VLinearSum()`, `N_VScale()`, `N_VProd()`, `N_VDiv()`, `N_VConst()`, `N_VClone()`, `N_VCloneVectorArray()`, `N_VDestroy()`, and `N_VDestroyVectorArray()`.

The nine files listed above can be extracted from the SUNDIALS `srcdir` and compiled by themselves into an SPGMR library or into a separate user code.

The following functions are available in the SPGMR package:

- `SpgmrMalloc`: allocates memory for `SpgmrSolve`;
- `SpgmrSolve`: solves $Ax = b$ using the SPGMR method;
- `SpgmrFree`: frees memory allocated by `SpgmrMalloc`.

The following functions are available in the support package `sundials_iterative.h` and `sundials_iterative.c`:

- `ModifiedGS`: performs the modified Gram-Schmidt orthogonalization procedure;
- `ClassicalGS`: performs the classical Gram-Schmidt orthogonalization procedure;
- `QRfact`: performs the QR factorization of a Hessenberg matrix;
- `QRsol`: solves a least squares problem with a Hessenberg matrix factored by `QRfact`.

7.2.2 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, but with `sundials_spbcgs.h` and `sundials_spbcgs.c` in place of `sundials_spgmr.h` and `sundials_spgmr.c`.

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocates memory for `SpbcgSolve`;

- `SpbcgSolve`: solves $Ax = b$ using the SPBCG method;
- `SpbcgFree`: frees memory allocated by `SpbcgMalloc`.

7.2.3 The SPTFQMR module

The SPTFQMR package, in the files `sundials_sptfqmr.h` and `sundials_sptfqmr.c`, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file `sundials_sptfqmr.h`.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, but with `sundials_sptfqmr.h` and `sundials_sptfqmr.c` in place of `sundials_spgmr.h` and `sundials_spgmr.c`.

The following functions are available in the SPTFQMR package:

- `SptfqmrMalloc`: allocates memory for `SptfqmrSolve`;
- `SptfqmrSolve`: solves $Ax = b$ using the SPTFQMR method;
- `SptfqmrFree`: frees memory allocated by `SptfqmrMalloc`.

7.2.4 The SPFGMR module

The SPFGMR package, in the files `sundials_spfgmr.h` and `sundials_spfgmr.c`, includes an implementation of the scaled preconditioned Flexible Generalized Minimum Residual method. For full details, including usage instructions, see the file `sundials_spfgmr.h`.

The files needed to use the SPFGMR module by itself are the same as for the SPGMR module, but with `sundials_spfgmr.h` and `sundials_spfgmr.c` in place of `sundials_spgmr.h` and `sundials_spgmr.c`.

The following functions are available in the SPFGMR package:

- `SpfgmrMalloc`: allocates memory for `SpfgmrSolve`;
- `SpfgmrSolve`: solves $Ax = b$ using the SPFGMR method;
- `SpfgmrFree`: frees memory allocated by `SpfgmrMalloc`.

7.2.5 The PCG module

The PCG package, in the files `sundials_pcg.h` and `sundials_pcg.c`, includes an implementation of the preconditioned conjugate gradient method. We note that due to the requirement of symmetric linear systems for the conjugate gradient method, this solver should only be used for problems with symmetric linear operators. Furthermore, aside from allowing a weight vector for computing weighted convergence norms, no variable or equation scaling is allowed for systems using this solver. For full details, including usage instructions, see the file `sundials_pcg.h`.

The files needed to use the PCG module by itself are the same as for the SPGMR module, but with `sundials_pcg.h` and `sundials_pcs.c` in place of `sundials_spgmr.h` and `sundials_spgmr.c`.

The following functions are available in the PCG package:

- `PcgMalloc`: allocates memory for `PcgSolve`;
- `PcgSolve`: solves $Ax = b$ using the PCG method;
- `PcgFree`: frees memory allocated by `PcgMalloc`.

7.3 Providing Alternate Linear Solver Modules

7.3.1 Newton system linear solver

The central ARKode module interfaces with the Newton system linear solver module using calls to one of four routines. These are denoted here by `linit()`, `lsetup()`, `lsolve()`, and `lfree()`. Briefly, their purposes are as follows:

- `linit()`: initializes and allocates memory specific to the linear solver;
- `lsetup()`: evaluates and preprocesses the Jacobian or preconditioner in preparation for solves;
- `lsolve()`: solves the linear system;
- `lfree()`: frees the linear solver memory.

A linear solver module must also provide a user-callable specification routine (like those described in the section *Linear solver specification functions*) which will attach the above four routines to the main ARKode memory block. The ARKode memory block is a structure defined in the header file `arkode_impl.h`. A pointer to such a structure is defined as the type `ARKodeMem`. The four fields in the `ARKodeMem` structure that refer to the Newton system linear solver's functions are `ark_linit`, `ark_lsetup`, `ark_lsolve`, and `ark_lfree`, respectively. Note that of these interface routines, only the `lsolve()` routine is required. The `lfree()` routine must be provided only if the solver specification routine makes any memory allocation. The linear solver specification function must also set the value of the field `ark_setupNonNull` in the ARKode memory block – to `TRUE` if `lsetup()` is used, or `FALSE` otherwise.

For consistency with the existing ARKode linear solver modules, we recommend that the return value of the specification function be 0 for a successful return or a negative value if an error occurs (e.g. if the pointer to the main ARKode memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, a memory allocation fails, etc).

To facilitate data exchange between the four interface functions, the field `ark_lmem` in the ARKode memory block can be used to attach a linear solver-specific memory block. That memory should be allocated in the linear solver specification function.

7.3.2 Mass matrix linear solver

Similarly, for problems involving a non-identity mass matrix $M \neq I$, the main ARKode module interfaces with the mass matrix linear solver module using calls to one of four routines: `minit()`, `msetup()`, `msolve()`, and `mfree()`. Briefly, their purposes are as follows:

- `minit()`: initializes and allocates memory specific to the mass matrix linear solver;
- `msetup()`: evaluates and preprocesses the mass matrix or associated preconditioner in preparation for solves;
- `msolve()`: solves the mass matrix system;
- `mfree()`: frees the mass matrix linear solver memory.

As with the Newton system linear solver, a mass matrix linear solver module must also provide a user-callable specification routine (like those described in the section *Linear solver specification functions*) which will attach the above four routines to the main ARKode memory block. The four fields in the `ARKodeMem` structure that refer to the mass matrix system linear solver's functions are `ark_minit`, `ark_msetup`, `ark_msolve`, and `ark_mfree`, respectively. As with the Newton system solver, only `msolve()` is required, and `mfree()` must be provided only if the solver specification routine makes any memory allocation. The mass matrix linear solver specification function must also set the value of the field `ark_MassSetupNonNull` in the ARKode memory block – to `TRUE` if `msetup()` is used, or `FALSE` otherwise.

For consistency with the existing ARKode linear solver modules, we recommend that the return value of the specification function be 0 for a successful return or a negative value if an error occurs (e.g. if the pointer to the main ARKode memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, a memory allocation fails, etc).

To facilitate data exchange between the four interface functions, the field `ark_mass_mem` in the ARKode memory block can be used to attach a linear solver-specific memory block. That memory should be allocated in the linear solver specification function.

These above routines that interface between ARKode and the Newton system or mass matrix linear solver module necessarily have fixed call sequences. Thus, a user wishing to implement another linear solver within the ARKode package must adhere to this set of interfaces. The following is a complete description of the call list for each of these routines. Note that the call list of each routine includes a pointer to the main ARKode memory block, by which the routine can access various data related to the ARKode solution. The contents of this memory block are given in the file `arkode_impl.h` (but not reproduced here, for the sake of space).

7.3.3 Initialization function

The type definition of `linit()` is

typedef int **(*linit)**(ARKodeMem *ark_mem*)

Completes initializations for the specific linear solver, such as counters and statistics.

Arguments:

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

Return value: Should return 0 if it has successfully initialized the ARKode linear solver and -1 otherwise.

Similarly, the type definition of `minit()` is

typedef int **(*minit)**(ARKodeMem *ark_mem*)

Completes initializations for the specific linear solver, such as counters and statistics.

Arguments:

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

Return value: Should return 0 if it has successfully initialized the ARKode linear solver and -1 otherwise.

7.3.4 Setup function

The type definition of `lsetup()` is

typedef int **(*lsetup)**(ARKodeMem *ark_mem*, int *convfail*, N_Vector *ypred*, N_Vector *fpred*, booleantype **jcurPtr*, N_Vector *vtemp1*, N_Vector *vtemp2*, N_Vector *vtemp3*)

Prepares the linear solver for subsequent calls to `lsolve()`. It may re-compute Jacobian-related data is it deems necessary.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *convfail* – an input flag used to indicate any problem that occurred during the solution of the nonlinear equation on the current time step for which the linear solver is being used. This flag can be used to help decide whether the Jacobian data kept by a linear solver needs to be updated or not. Its possible values are:
  - *ARK_NO_FAILURES*: this value is passed if either this is the first call for this step, or the local error test failed on the previous attempt at this step (but the Newton iteration converged).

  - *ARK_FAIL_BAD_J*: this value is passed if (a) the previous Newton corrector iteration did not converge and the linear solver's setup routine indicated that its Jacobian-related data is not current, or (b) during the previous Newton corrector iteration, the linear solver's solve routine failed in a recoverable manner and the linear solver's setup routine indicated that its Jacobian-related data is not current.
  - *ARK_FAIL_OTHER*: this value is passed if during the current internal step try, the previous Newton iteration failed to converge even though the linear solver was using current Jacobian-related data.
- *ypred* – is the predicted y vector for the current ARKode internal step.
- *fpred* – is the value of the implicit right-hand side at *ypred*, i.e. $f_I(t_n, ypred)$.
- *jcurPtr* – is a pointer to a boolean to be filled in by `lsetup()`. The function should set `*jcurPtr = TRUE` if its Jacobian data is current after the call and should set `*jcurPtr = FALSE` if its Jacobian data is not current. If `lsetup()` calls for re-evaluation of Jacobian data (based on *convfail* and ARKode state data), it should return `*jcurPtr = TRUE` unconditionally; otherwise an infinite loop can result.
- *vtemp1*, *vtemp2*, *vtemp3* – are temporary variables of type `N_Vector` provided for use by `lsetup()`.

Return value: Should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Similarly, the type definition of `msetup()` is

typedef int **(*msetup)** (ARKodeMem *ark_mem*, N_Vector *vtemp1*, N_Vector *vtemp2*, N_Vector *vtemp3*)

Prepares the mass matrix linear solver for subsequent calls to `msolve()`. It may re-compute mass-matrix-related data is it deems necessary.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *vtemp1*, *vtemp2*, *vtemp3* – are temporary variables of type `N_Vector` provided for use by `msetup()`.

Return value: Should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

7.3.5 Solve function

The type definition of `lsolve()` is

typedef int **(*lsolve)** (ARKodeMem *ark_mem*, N_Vector *b*, N_Vector *weight*, N_Vector *ycur*, N_Vector *fcur*)

Solves the linear equation $\mathcal{A}x = b$, where $\mathcal{A}$ arises in the Newton iteration (see the section *Linear solver methods*) and gives some approximation to $M - \gamma J$, $J = \frac{\partial}{\partial y} f_I(t_n, ycur)$. Note, the right-hand side vector b is input, and γ is available as `ark_mem->ark_gamma`.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *b* – is the right-hand side vector b. The solution is also to be returned in the vector b.
- *weight* – is a vector that contains the residual weights. These are the rwt_i of *Residual weight function*.
- *ycur* – is a vector that contains the solver's current approximation to $y(t_n)$.
- *fcur* – is a vector that contains $f_I(t_n, ycur)$.

Return value: Should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Similarly, the type definition of `msolve()` is

typedef int **(*msolve)**(ARKodeMem *ark_mem*, N_Vector *b*, N_Vector *weight*)

Solves the linear equation $Mx = b$, where M is the system mass matrix. Note, the right-hand side vector b is input, and holds the solution x on output.

Arguments:

- *arkode_mem* – pointer to the ARKode memory block.
- *b* – is the right-hand side vector b. The solution is also to be returned in the vector b.
- *weight* – is a vector that contains the error weights. These are the rwt_i of *Residual weight function*.

Return value: Should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

7.3.6 Memory deallocation function

The type definition of `lfree()` is

typedef void **(*lfree)**(ARKodeMem *ark_mem*)

free up any memory allocated by the linear solver.

Arguments:

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

Return value: None

Notes: This routine is called once a problem has been completed and the linear solver is no longer needed.

Similarly, the type definition of `mfree()` is

typedef void **(*mfree)**(ARKodeMem *ark_mem*)

free up any memory allocated by the mass matrix linear solver.

Arguments:

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

Return value: None

Notes: This routine is called once a problem has been completed and the mass matrix solver is no longer needed.

CHAPTER

EIGHT

ARKODE INSTALLATION PROCEDURE

The installation of ARKode is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The installation procedure remains the same, whether or not the downloaded file contains solvers other than ARKode. [1]

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form `SOLVER-X.Y.Z.tar.gz`, where `SOLVER` is one of: `sundials`, `arkode`, `cvode`, `cvodes`, `ida`, `idas`, or `kinsol`, and `X.Y.Z` represents the version number (of the SUNDIALS suite or of the individual solver). To begin the installation, first uncompress and expand the sources, by issuing

```
% tar -zxf SOLVER-X.Y.Z.tar.gz
```

This will extract source files under a directory `SOLVER-X.Y.Z`.

Starting with version 2.4.0 of SUNDIALS, two installation methods are provided: a standard LINUX/UNIX autotools-based method (see the section *Autotools-based installation*), and a new method based on CMake (see the section *CMake-based installation*). However, before providing detailed explanations on the installation procedure for the two approaches, we begin with a few common observations:

- In the remainder of this chapter, we make the following distinctions:

  `SRCDIR` is the directory `SOLVER-X.Y.Z` created above; i.e. the directory containing the SUNDIALS sources.

  `BUILDDIR` is the (temporary) directory under which SUNDIALS is built.

  `INSTDIR` is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory `INSTDIR/include` while libraries are installed under `INSTDIR/lib`, with `INSTDIR` specified at configuration time.

- For the CMake-based installation, in-source builds are prohibited; in other words, the build directory `BUILDDIR` can **not** be the same as `SRCDIR` and such an attempt will lead to an error. For autotools-based installation, in-source builds are allowed, although even in that case we recommend using a separate `BUILDDIR`. Indeed, this prevents "polluting" the source tree and allows efficient builds for different configurations and/or options.

[1] Files for both the serial and parallel versions of ARKode are included in the distribution. For users in a serial computing environment, the files specific to parallel environments (which may be deleted) are as follows:

- all files in `src/nvec_par/`;
- `nvector parallel.h` (in `include/nvector/`);
- `arkode_bbdpre.c`, `arkode_bbdpre_impl.h` (in `src/arkode/`);
- `arkode_bbdpre.h` (in `include/arkode/`);
- `farkbbd.c`, `farkbbd.h` (in `src/arkode/fcmix/`);
- all files in `examples/arkode/parallel/`;
- all files in `examples/arkode/fcmix_parallel/`.

(By "serial version" of ARKode we mean the ARKode solver with the serial NVECTOR module attached, and similarly for "parallel version" we mean the ARKode solver with the parallel NVECTOR module attached.)

- For both methods, the installation directory `INSTDIR` can not be the same as the source directory `SRCDIR`.
- By default, only the libraries and header files are exported to the installation directory `INSTDIR`. If enabled by the user (with the appropriate option to `configure` or toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries. This installation will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the installed SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as "templates" for your own problems. The `configure` script will install makefiles. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) makefiles. Note that both installation approaches also allow the option of building the SUNDIALS examples without having to install them (useful as a sanity check for the freshly built libraries).
- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.

Further details on the autotools- and CMake-based installation procedures, instructions for manual compilation, and a roadmap of the resulting installed libraries and exported header files, are provided in the following subsections:

- *Autotools-based installation*
- *CMake-based installation*
- *Manually building SUNDIALS*
- *Installed libraries and exported header files*

8.1 Autotools-based installation

The installation procedure outlined below will work on commodity LINUX/UNIX systems without modification. However, users are still encouraged to carefully read this entire section before attempting to install the SUNDIALS suite, in case non-default choices are desired for compilers, compilation options, installation location, etc. The user may invoke the configuration script with the "help" flag to view a complete listing of available options, by issuing the command

```
$ ./configure --help
```

from within `SRCDIR`.

Note: In the command arguments within this section, the preceding "$" refers to the LINUX/UNIX command prompt, and should not be entered within each command.

The installation steps for SUNDIALS can be as simple as the following:

```
$ cd SRCDIR
$ ./configure
$ make
$ make install
```

in which case the SUNDIALS header files and libraries are installed under `/usr/local/include` and `/usr/local/lib`, respectively. Note that, by default, the example programs are not built and installed. To subsequently delete all temporary files created by building SUNDIALS, issue

```
$ make clean
```

To prepare the SUNDIALS distribution for a new install (using, for example, different options and/or installation destinations), issue

```
$ make distclean
```

We note that the above steps are for an "in-source" build (not recommended). For an "out-of-source" build (recommended), the procedure is simply:

```
$ cd BUILDDIR
$ SRCDIR/configure
$ make
$ make install
```

Note that, in this case, `make clean` and `make distclean` are irrelevant. Indeed, if disk space is a priority, the entire `BUILDDIR` can be purged after the installation completes. For a new install, a new `BUILDDIR` directory can be created and used.

8.1.1 Configuration options

The installation procedure given above will generally work without modification; however, if the system includes multiple (or non-default) MPI implementations, then certain configure script-related options may be used to indicate which MPI implementation should be used. Also, if the user wants to use non-default language compilers, then, again, the necessary shell environment variables must be appropriately redefined. The remainder of this section provides explanations of available configure script options.

General options

–prefix=PREFIX Location for architecture-independent files.

Default: `PREFIX=/usr/local`

–exec-prefix=EPREFIX Location for architecture-dependent files.

Default: `EPREFIX=/usr/local`

–includedir=DIR Alternate location for installation of header files.

Default: `DIR=PREFIX/include`

–libdir=DIR Alternate location for installation of libraries.

Default: `DIR=EPREFIX/lib`

–disable-solver Although each existing solver module is built by default, support for a given solver can be explicitly disabled using this option. The valid values for solver are: `arkode`, `cvode`, `cvodes`, `ida`, `idas`, and `kinsol`. For example,

```
$ ./configure --disable-cvodes --disable-idas
```

–enable-examples Available example programs are not built by default. Use this option to enable compilation of all pertinent example programs. Upon completion of the `make` command, the example executables will be created under solver-specific subdirectories of `BUILDDIR/examples`, e.g.:

`BUILDDIR/examples/SOLVER/serial`: serial C examples

`BUILDDIR/examples/SOLVER/parallel`: parallel C examples

`BUILDDIR/examples/SOLVER/fcmix_serial`: serial Fortran examples

`BUILDDIR/examples/SOLVER/fcmix_parallel`: parallel Fortran examples

Note: Some of these subdirectories may not exist depending upon the solver and/or the configuration options given.

Note: Since ARKode is packaged with examples in C, C++, Fortran 77 and Fortran 90, the ARKode examples are installed in the subdirectories:

`BUILDDIR/examples/arkode/C_serial`: serial C examples

`BUILDDIR/examples/SOLVER/C_parallel`: parallel C examples

`BUILDDIR/examples/SOLVER/F77_serial`: serial Fortran 77 examples

`BUILDDIR/examples/SOLVER/F77_parallel`: parallel Fortran 77 examples

At present, the SUNDIALS Autotools-based build system does not install ARKode's C++ or Fortran 90 examples.

–with-examples-instdir=EXINSTDIR Alternate location for example executables and sample output files (valid only if examples are enabled). Note that installation of example files can be completely disabled by issuing `EXINSTDIR=no` (in case building the examples is desired only as a test of the SUNDIALS libraries).

Default: `DIR=EPREFIX/examples`

–with-cppflags=ARG Specify additional C preprocessor flags (e.g., `--with-cppflags=-I/usr/local/include`>) if necessary header files are located in nonstandard locations.

–with-cflags=ARG Specify additional C compilation flags.

–with-ldflags=ARG Specify additional linker flags (e.g., `--with-ldflags=-L/usr/local/lib`) if required libraries are located in nonstandard locations.

–with-libs=ARG Specify additional libraries to be used (e.g., `--with-libs=-lfoo` to link with the library named `libfoo.a` or `libfoo.so`).

–with-precision=ARG By default, SUNDIALS will define a real number (internally referred to as `realtype`) to be a double-precision floating-point numeric data type (`double` C-type); however, this option may be used to build SUNDIALS with an alternate `realtype`:

`--with-precision=single` declares `realtype` as a single-precision floating-point numeric data type (`float` C-type).

`--with-precision=extended` declares `realtype` as a double-precision floating-point numeric data type (`long double` C-type).

Default `double`:

Users should not build SUNDIALS with support for single-precision floating-point arithmetic on 32- or 64-bit systems. This will almost certainly result in unreliable numerical solutions. The configuration option `--with-precision=single` is intended for systems on which single-precision arithmetic involves at least 14 decimal digits.

Options for Fortran support

–disable-fcmix Using this option will disable all Fortran support. The FARKODE, FCVODE, FKINSOL, FIDA and FNVECTOR modules will not be built, regardless of availability.

–with-fflags=ARG Specify additional Fortran compilation flags.

Options for MPI support

The following configuration options are only applicable to the parallel SUNDIALS packages:

–disable-mpi Using this option will completely disable MPI support.

–with-mpicc=ARG By default, the configuration utility script will use the MPI compiler script named `mpicc` to compile the C-language parallelized SUNDIALS subroutines; however, for reasons of compatibility, different executable names may be specified.

Also, `--with-mpicc=no` can be used to disable the use of MPI compiler scripts, thus causing the serial C compilers to be used to compile the parallelized SUNDIALS functions and examples.

–with-mpif77=ARG As described above for C routines, the configuration utility script will use the MPI compiler script named `mpif77` to compile the Fortran parallelized SUNDIALS subroutines. Similarly, `--with-mpif77=no` can be used to force the serial Fortran compiler to compile the parallelized SUNDIALS functions and examples.

–with-mpi-root=MPIDIR This option may be used to specify which MPI implementation should be used. The SUNDIALS configuration script will automatically check under the subdirectories `MPIDIR/include` and `MPIDIR/lib` for the necessary header files and libraries. The subdirectory `MPIDIR/bin` will also be searched for the C and Fortran MPI compiler scripts, unless the user specifies `--with-mpicc=no` or `--with-mpif77=no`.

–with-mpi-flags=ARG Specify additional MPI-specific compilation flags.

If a user would prefer not to use a preexisting MPI compiler script, but instead use a serial compiler and provide the flags necessary to compile the MPI-aware subroutines in SUNDIALS, the following options may be used.

–with-mpi-incdir=INCDIR Include directory for MPI library; must include `mpi.h`.

Default: `INCDIR=MPIDIR/include`

–with-mpi-libdir=LIBDIR Library directory for MPI library.

Default: `LIBDIR=MPIDIR/lib`

–with-mpi-libs=LIBS MPI library files to link with example executables (e.g., `--with-mpi-libs=-lmpich`).

Options for library support

By default, only static libraries are built, but the following option may be used to build shared libraries on supported platforms.

–enable-shared Using this particular option will result in both static and shared versions of the available SUNDIALS libraries being built if the system supports shared libraries. To build only shared libraries also specify `--disable-static`.

Note: The FARKODE, FCVODE, FKINSOL and FIDA libraries can only be built as static libraries because they contain references to externally defined symbols, namely user-supplied Fortran subroutines. Although the Fortran interfaces to the serial and parallel implementations of the supplied NVECTOR module do not contain any unresolvable external symbols, the libraries are still built as static libraries for the purpose of consistency.

Options for BLAS/LAPACK support

The `configure` script will attempt to automatically determine the proper libraries to be linked for support of the BLAS/LAPACK linear solver module. If these are not found, or if BLAS and/or LAPACK libraries are installed in a non-standard location, the following options can be used:

–with-blas=BLASDIR Specify the BLAS library.

Default: none

–with-lapack=LAPACKDIR Specify the LAPACK library.

Default: none

Environment variables

The following environment variables can be locally (re)defined for use during the configuration of SUNDIALS. See the next section for illustrations of these.

CC Since the configuration script uses the first C compiler found in the current executable search path, then `CC` must be locally (re)defined in order to use a different compiler. For example, to use `xcc` (executable name of chosen compiler) as the C language compiler,

```
$ ./configure CC=xcc
```

F77 As above, since the configuration script uses the first Fortran compiler found in the current executable search path, then `F77` must be locally (re)defined in order to use a different compiler.

CFLAGS Overrides the default C compilation flags.

FFLAGS Overrides the default Fortran compilation flags.

8.1.2 Configuration examples

The following examples are meant to help demonstrate proper usage of the configure options.

In each example, we assume that the SUNDIALS source files are in the subdirectory `/home/myname/sundials_src/`, and that the library is being built in the directory `/home/myname/sundials_build/`.

To build SUNDIALS using the default C and Fortran compilers, the default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under appropriate subdirectories of `/home/myname/sundials/`, use

```
$ cd /home/myname/sundials_build
$ /home/myname/sundials_src/configure --prefix=/home/myname/sundials --enable-examples
```

To disable installation of the examples, use:

```
$ cd /home/myname/sundials_build
$ /home/myname/sundials_src/configure --prefix=/home/myname/sundials \
          --enable-examples --with-examples-instdir=no
```

The following example builds SUNDIALS using `gcc` as the serial C compiler, `gfortran` as the serial Fortran compiler, the default `mpicc` as the parallel C compiler, the default `mpif77` as the parallel Fortran compiler, and appends the `-O3` compilaton flag to the list of default flags:

```
$ cd /home/myname/sundials_build
$ /home/myname/sundials_src/configure CC=gcc F77=gfortran --with-cflags=-O3 \
          --with-fflags=-O3 --with-mpicc=mpicc --with-mpif77=mpif77
```

The next example again builds SUNDIALS using `gcc` as the serial C compiler, but the `--with-mpicc=no` option explicitly disables the use of the corresponding MPI compiler script. In addition, since the `--with-mpi-root` option is given, the compilation flags `-I/usr/apps/mpich/1.2.4/include` and `-L/usr/apps/mpich/1.2.4/lib` are passed to `gcc` when compiling the MPI-enabled functions. The `--with-mpi-libs` option is required so that the configure script can check if `gcc` can link with the appropriate MPI library. The `--disable-lapack` option explicitly disables support for BLAS/LAPACK, while the

--disable-fcmix explicitly disables building the FCMIX interfaces. Note that, because of the last two options, no Fortran-related settings are checked for.

```
$ cd /home/myname/sundials_build
$ /home/myname/sundials_src/configure CC=gcc --with-mpicc=no \
            --with-mpi-root=/usr/apps/mpich/1.2.4 \
            --with-mpi-libs=-lmpich \
            --disable-lapack --disable-fcmix
```

Finally, a minimal configuration and installation of SUNDIALS in /home/myname/sundials/ (serial only, no Fortran support, no examples) can be obtained with:

```
$ cd /home/myname/sundials_build
$ /home/myname/sundials_src/configure --prefix=/home/myname/sundials \
            --disable-mpi --disable-lapack --disable-fcmix
```

8.2 CMake-based installation

Support for CMake-based installation has been added to SUNDIALS primarily to provide a platform-independent build system. Like autotools, CMake can generate a Unix Makefile. Unlike autotools, CMake can also create KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration files. In addition, CMake provides a GUI front end, allowing a more interactive installation process than when using autotools.

The installation options are very similar to the options mentioned above (although their default values may differ slightly). Practically, all configurations supported by the autotools-based installation approach are also possible with CMake, the only notable exception being cross-compilation, which is currently not implemented in the CMake approach.

However, the CMake approach does support installation of the C++ and Fortran 90 examples supplied with ARKode (both serial and parallel), allowing for a complete installation of all possible example programs supplied with SUNDIALS.

The SUNDIALS build process requires CMake version 2.4.x or higher and a working compiler. On Unix-like operating systems, it also requires Make (and curses, including its development libraries, for the GUI front end to CMake, ccmake or cmake-gui), while on Windows it requires Visual Studio. While many Linux distributions offer CMake, the version included may be out of date and should be verified. Many new CMake features have been added recently, and it is recommended that you download the latest version from http://www.cmake.org/HTML/Download.html. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use ccmake or cmake-gui (depending the version of CMake), while Windows users will be able to use CMakeSetup.

As noted above, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a make distclean procedure and it is therefore difficult to clean-up the source tree after an in-source build).

8.2.1 Configuring, building, and installing on Unix-like systems

These instructions use ccmake from the CMake installed location. ccmake is a Curses based GUI for CMake. For newer versions of CMake, you may replace each ccmake command with cmake-gui, that provides a full point-and-click graphical user interface to CMake. When using cmake-gui, make sure to select the appropriate source and the build directory. Also, make sure to pick the “Unix Makefiles” generator. Some CMake versions will ask you to select the generator the first time you press Configure instead of having a drop-down menu in the main dialog. All remaining instructions should follow similarly.

To run `ccmake`, go to the build directory and specify as an argument the source directory:

```
$ mkdir BUILDDIR
$ cd BUILDDIR
$ ccmake SRCDIR
```

About `ccmake`:

- Iterative process
  - Select values, run configure (press the `<c>` key)
  - Set the settings, run configure, set the settings, run configure, etc.
- Repeat until all values are set and the generate option is available (press the `<g>` key)
- Some variables (advanced variables) are not visible right away
- To see advanced varables, toggle to advanced mode (press the `<t>` key)
- To set a variable, move the cursor to the variable and press `<enter>`
  - If it is a boolean (`ON/OFF`) it will flip the value
  - If it is string or file, it will allow editing of the string
  - For file and directories, the `<tab>` key can be used to auto-complete
  - (`cmake-gui` only) If it is file or directory, an ellipsis button will appear (”...”) on the far right of the entry. Clicking this button will bring up the file or directory selection dialog.
- To search for a variable press the `</>` key, and to repeat the search, press the `<n>` key

CMake will now generate makefiles including all dependencies and all rules to build SUNDIALS on this system. You should not, however, try to move the build directory to another location on this system or to another system. Once you have makefiles you should be able to just type:

```
$ make
```

To install SUNDIALS in the installation directory specified at configuration time, simply run

```
$ make install
```

8.2.2 Configuring, building, and installing on Windows

These instructions use CMakeSetup from the CMake install location. Make sure to select the appropriate source and the build directory. Also, make sure to pick the appropriate generator (e.g. on Visual Studio 6, pick the Visual Studio 6 generator). Some CMake versions will ask you to select the generator the first time you press Configure instead of having a drop-down menu in the main dialog.

About `CMakeSetup`:

- Iterative process
  - Select values, press the Configure button
  - Set the settings, run configure, set the settings, run configure, etc.
- Repeat until all values are set and the `OK` button becomes available.
- Some variables (advanced variables) are not visible right away
- To see advanced varables, toggle to advanced mode (“Show Advanced Values” toggle).
- To set the value of a variable, click on that value.

- If it is boolean (`ON/OFF`), a drop-down menu will appear for changing the value.
- If it is file or directory, an ellipsis button will appear ("...") on the far right of the entry. Clicking this button will bring up the file or directory selection dialog.
- If it is a string, it will become an editable string.

CMake will now create Visual Studio project files. You should now be able to open the SUNDIALS project (or workspace) file. Make sure to select the appropriate build type (Debug, Release, ...). To build SUNDIALS, simply build the `ALL_BUILD` target. To install SUNDIALS, simply run the `INSTALL` target within the build system.

8.2.3 Configuration options

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only. Some of them will be different on different systems.

BUILD_ARKODE Build the ARKODE library

Default: `ON`

BUILD_CVODE Build the CVODE library

Default: `ON`

BUILD_CVODES Build the CVODES library

Default: `ON`

BUILD_IDA Build the IDA library

Default: `ON`

BUILD_IDAS Build the IDAS library

Default: `ON`

BUILD_KINSOL Build the KINSOL library

Default: `ON`

BUILD_SHARED_LIBS Build shared libraries

Default: `OFF`

BUILD_STATIC_LIBS Build static libraries

Default: `ON`

CMAKE_BUILD_TYPE Choose the type of build, options are: `None` (`CMAKE_C_FLAGS` used), `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel`

Default:

CMAKE_CXX_COMPILER C++ compiler

Default: `/usr/bin/g++`

CMAKE_CXX_FLAGS Flags for C++ compiler

Default:

CMAKE_CXX_FLAGS_DEBUG Flags used by the C++ compiler during debug builds

Default: `-g`

CMAKE_CXX_FLAGS_MINSIZEREL Flags used by the C++ compiler during release minsize builds

Default: `-Os -DNDEBUG`

CMAKE_CXX_FLAGS_RELEASE Flags used by the C++ compiler during release builds

Default: `-O3 -DNDEBUG`

CMAKE_CXX_FLAGS_RELWITHDEBINFO Flags used by the C++ compiler during release builds (with debugging enabled)

Default: `-O2 -g`

CMAKE_C_COMPILER C compiler

Default: `/usr/bin/gcc`

CMAKE_C_FLAGS Flags for C compiler

Default:

CMAKE_C_FLAGS_DEBUG Flags used by the compiler during debug builds

Default: `-g`

CMAKE_C_FLAGS_MINSIZEREL Flags used by the compiler during release minsize builds

Default: `-Os -DNDEBUG`

CMAKE_C_FLAGS_RELEASE Flags used by the compiler during release builds

Default: `-O3 -DNDEBUG`

CMAKE_C_FLAGS_RELWITHDEBINFO Flags used by the C compiler during release builds (with debugging enabled)

Default: `-O2 -g`

CMAKE_BACKWARDS_COMPATIBILITY For backwards compatibility, what version of CMake commands and syntax should this version of CMake allow.

Default: `2.4`

CMAKE_Fortran_COMPILER Fortran compiler

Default: `/usr/bin/gfortran`

Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (`FCMIX_ENABLE` is `ON`) or BLAS/LAPACK support is enabled (`LAPACK_ENABLE` is `ON`).

CMAKE_Fortran_FLAGS Flags for Fortran compiler

Default:

CMAKE_Fortran_FLAGS_DEBUG Flags used by the Fortran compiler during debug builds

Default: `-g`

CMAKE_Fortran_FLAGS_MINSIZEREL Flags used by the Fortran compiler during release minsize builds

Default: `-Os`

CMAKE_Fortran_FLAGS_RELEASE Flags used by the Fortran compiler during release builds

Default: `-O3`

CMAKE_Fortran_FLAGS_RELWITHDEBINFO Flags used by the Fortran compiler during release builds (with debugging enabled)

Default: `-O2 -g`

CMAKE_INSTALL_PREFIX Install path prefix, prepended onto install directories

Default: `/usr/local`

Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories `include` and `lib` of `CMAKE_INSTALL_PREFIX`, respectively.

CXX_ENABLE Flag to enable C++ ARKode examples (if examples are enabled)

Default: `OFF`

EXAMPLES_ENABLE Build the SUNDIALS examples

Default: `OFF`

Note: setting this option to `ON` will trigger additional options related to how and where example programs will be installed.

EXAMPLES_GENERATE_MAKEFILES Create Makefiles for building the examples

Default: `ON`

Note: This option is triggered only if enabling the building and installing of the example programs (i.e., both `EXAMPLES_ENABLE` and `EXAMPLEs_INSTALL` are set to `ON`) and if configuration is done on a Unix-like system. If enabled, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by `EXAMPLES_INSTALL_PATH`.

EXAMPLES_INSTALL Install example files

Default: `ON`

Note: This option is triggered only if building example programs is enabled (`EXAMPLES_ENABLE` is set to `ON`). If the user requires installation of example programs then the sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by `EXAMPLES_INSTALL_PATH`. A CMake configuration script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, an additional option (`EXAMPLES_GENERATE_MAKEFILES`) will be triggered.

EXAMPLES_INSTALL_PATH Output directory for installing example files

Default: `/usr/local/examples`

Note: The actual default value for this option will be an `examples` subdirectory created under `CMAKE_INSTALL_PREFIX`.

EXAMPLES_USE_STATIC_LIBS Link examples using the static libraries

Default: `OFF`

Note: This option is triggered only if building shared libraries is enabled (`BUILD_SHARED_LIBS` is `ON`).

F90_ENABLE Flag to enable Fortran 90 ARKode examples (if examples are enabled)

Default: `OFF`

FCMIX_ENABLE Enable Fortran-C support

Default: `OFF`

LAPACK_ENABLE Enable LAPACK support

Default: `OFF`

Note: Setting this option to `ON` will trigger the two additional options see below.

LAPACK_LIBRARIES LAPACK (and BLAS) libraries

Default: `/usr/lib/liblapack.so;/usr/lib/libblas.so`

LAPACK_LINKER_FLAGS LAPACK (and BLAS) required linker flags

Default: `-lg2c`

MPI_ENABLE Enable MPI support

Default: `OFF`

Note: Setting this option to `ON` will trigger several additional options related to MPI.

MPI_MPICC `mpicc` program

Default: `/home/radu/apps/mpich1/gcc/bin/mpicc`

Note: This option is triggered only if using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is `ON`).

MPI_MPICXX `mpicxx` program

Default:

Note: This option is triggered only if using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is `ON`) and C++ is enabled (`CXX_ENABLE` is `ON`).

MPI_MPIF77 `mpif77` program

Default: `/home/radu/apps/mpich1/gcc/bin/mpif77`

Note: This option is triggered only if using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is `ON`) and Fortran-C support is enabled (`FCMIX_ENABLE` is `ON`).

MPI_MPIF90 `mpif90` program

Default:

Note: This option is triggered only if using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is `ON`), Fortran-C support is enabled (`FCMIX_ENABLE` is `ON`), and Fortran 90 examples are enabled (`F90_ENABLE` is `ON`).

MPI_INCLUDE_PATH Path to MPI header files

Default: `/home/radu/apps/mpich1/gcc/include`

Note: This option is triggered only if not using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is `OFF`).

MPI_LIBRARIES MPI libraries

Default: `/home/radu/apps/mpich1/gcc/lib/libmpich.a`

Note: This option is triggered only if not using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is `OFF`).

MPI_USE_MPISCRIPTS Use MPI compiler scripts

Default: `ON`

SUNDIALS_PRECISION Precision used in SUNDIALS, options are: `double`, `single` or `extended`

Default: `double`

USE_GENERIC_MATH Use generic (`stdc`) math libraries

Default: `ON`

8.3 Manually building SUNDIALS

With the addition of CMake support, the installation of the SUNDIALS package on almost any platform was greatly simplified. However, if for whatever reason, neither of the two procedures described above is convenient (for example for users who prefer to own the build process or otherwise incorporate SUNDIALS or one of its solvers in a larger project with its own build system), we provide a few directions for a completely manual installation.

The following files are required to compile a SUNDIALS solver module:

- public header files are located under `SRCDIR/include/SOLVER`
- implementation header files and source files are located under `SRCDIR/src/SOLVER`
- (optional) Fortran/C interface files are located under `SRCDIR/src/SOLVER/fcmix`
- shared public header files are located under `SRCDIR/include/sundials`
- shared source files are located under `SRCDIR/src/sundials`
- (optional) NVECTOR_SERIAL header and source files are located under `SRCDIR/include/nvector` and `SRCDIR/src/nvec_ser`
- (optional) NVECTOR_PARALLEL header and source are files located under `SRCDIR/include/nvector` and `SRCDIR/src/nvec_par`
- the configuration header file, `sundials_config.h` (see below)

A sample header file that, appropriately modified, can be used as `sundials_config.h` (otherwise created automatically by the configure or CMake scripts), is provided below.

```
/* SUNDIALS configuration header file */
#define SUNDIALS_PACKAGE_VERSION "2.5.0"

#define SUNDIALS_F77_FUNC(name,NAME) name ## _
#define SUNDIALS_F77_FUNC_(name,NAME) name ## _

#define SUNDIALS_DOUBLE_PRECISION 1

#define SUNDIALS_USE_GENERIC_MATH

#define SUNDIALS_BLAS_LAPACK 1

#define SUNDIALS_MPI_COMM_F2C 1
```

```
#define SUNDIALS_EXPORT
```

The various preprocessor macros defined within `sundials_config.h` have the following uses:

- Precision of the SUNDIALS `realtype` type

  Only one of the macros SUNDIALS_SINGLE_PRECISION, SUNDIALS_DOUBLE_PRECISION and SUNDIALS_EXTENDED_PRECISION should be defined to indicate if the SUNDIALS `realtype` type is an alias for `float`, `double`, or `long double`, respectively.

- Use of generic math functions

  If SUNDIALS_USE_GENERIC_MATH is defined, then the functions in `sundials_math.h` and `sundials_math.c` will use the `pow`, `sqrt`, `fabs`, and `exp` functions from the standard math library (see `math.h`), regardless of the definition of `realtype`. Otherwise, if `realtype` is defined to be an alias for the `float` C-type, then SUNDIALS will use `powf`, `sqrtf`, `fabsf`, and `expf`. If `realtype` is instead defined to be a synonym for the `long double` C-type, then `powl`, `sqrtl`, `fabsl`, and `expl` will be used.

  Note: Although the `powf/powl`, `sqrtf/sqrtl`, `fabsf/fabsl`, and `expf/expl` routines are not specified in the ANSI C standard, they are ISO C99 requirements. Consequently, these routines will only be used if available.

- Fortran name-mangling scheme

  The macros given below are used to transform the C-language function names defined in the Fortran-C interface modules in a manner consistent with the preferred Fortran compiler, thus allowing native C functions to be called from within a Fortran subroutine. The name-mangling scheme is specified by appropriately defining the following parameterized macros (using the stringization operator, `##`, if necessary):

  – SUNDIALS_F77_FUNC(name,NAME)

  – SUNDIALS_F77_FUNC_(name,NAME)

  For example, to specify that mangled C-language function names should be lowercase with two underscores appended, include

  ```
  #define SUNDIALS_F77_FUNC(name,NAME) name ## __
  #define SUNDIALS_F77_FUNC_(name,NAME) name ## __
  ```

  in the `sundials_config.h` header file.

- Availability of BLAS/LAPACK libraries

  If working libraries for BLAS and LAPACK are available, then the macro SUNDIALS_BLAS_LAPACK should be set to 1; otherwise it should have the value 0.

- Use of an MPI communicator other than `MPI_COMM_WORLD` in Fortran

  If the macro SUNDIALS_MPI_COMM_F2C is defined, then the MPI implementation used to build SUNDIALS defines the type `MPI_Fint` and the function `MPI_Comm_f2c`, and it is possible to use MPI communicators other than `MPI_COMM_WORLD` with the Fortran-C interface modules.

- The macro SUNDIALS_EXPORT is used when marking SUNDIALS API functions for export/import. When building shared SUNDIALS libraries under Windows, use

  ```
  #define SUNDIALS_EXPORT __declspec(dllexport)
  ```

  When linking to shared SUNDIALS libraries under Windows, use

  ```
  #define SUNDIALS_EXPORT __declspec(dllimport)
  ```

  In all other cases (other platforms or static libraries under Windows), the `SUNDIALS_EXPORT` macro is empty.

8.4 Installed libraries and exported header files

Using the standard SUNDIALS build system, the command

```
$ make install
```

will install the libraries under `LIBDIR` and the public header files under `INCLUDEDIR`. The default values for these directories are `INSTDIR/lib` and `INSTDIR/include`, respectively, but can be changed using the configure script options `--prefix`, `--exec-prefix`, `--includedir` and `--libdir` (see the section *Autotools-based installation*) or the appropriate CMake options (see the section *CMake-based installation*). For example, a global installation of SUNDIALS on a LINUX/UNIX system to the system-level directory `/opt/sundials-2.5.0` could be accomplished using

```
$ configure --prefix=/opt/sundials-2.5.0
```

Although all installed libraries reside under `LIBDIR`, the public header files are further organized into subdirectories under `INCLUDEDIR`.

The installed libraries and exported header files are listed for reference in the *Table: SUNDIALS libraries and header files*. The file extension `.LIB` is typically `.so` for shared libraries and `.a` for static libraries. Note that, in this table names are relative to `LIBDIR` for libraries and to `INCLUDEDIR` for header files.

A typical user program need not explicitly include any of the shared SUNDIALS header files from under the `INCLUDEDIR/sundials` directory since they are explicitly included by the appropriate solver header files (e.g., `arkode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so (e.g., the functions declared in `sundials_dense.h` could be used in building a preconditioner).

8.4.1 Table: SUNDIALS libraries and header files

Shared	Header files	`sundials/sundials_band.h`, `sundials/sundials_config.h`, `sundials/sundials_dense.h`, `sundials/sundials_direct.h`, `sundials/sundials_fnvector.h`, `sundials/sundials_iterative.h`, `sundials/sundials_lapack.h`, `sundials/sundials_math.h`, `sundials/sundials_nvector.h`, `sundials/sundials_pcg.h`, `sundials/sundials_spbcgs.h`, `sundials/sundials_spfgmr.h`, `sundials/sundials_spgmr.h`, `sundials/sundials_sptfqmr.h`, `sundials/sundials_types.h`
Serial NVECTOR	Libraries	`libsundials_nvecserial.LIB`, `libsundials_fnvecserial.a`
Serial NVECTOR	Header files	`nvector/nvector_serial.h`
Parallel NVECTOR	Libraries	`libsundials_nvecparallel.LIB`, `libsundials_fnvecparallel.a`
Parallel NVECTOR	Header files	`nvector/nvector_parallel.h`
ARKODE	Libraries	`libsundials_arkode.LIB`, `libsundials_farkode.a`
ARKODE	Header files	`arkode/arkode.h`, `arkode/arkode_band.h`, `arkode/arkode_bandpre.h`, `arkode/arkode_bbdpre.h`, `arkode/arkode_dense.h`, `arkode/arkode_direct.h`, `arkode/arkode_impl.h`, `arkode/arkode_lapack.h`, `arkode/arkode_pcg.h`, `arkode/arkode_spbcgs.h`, `arkode/arkode_spfgmr.h`, `arkode/arkode_spgmr.h`, `arkode/arkode_spils.h`, `arkode/arkode_sptfqmr.h`
CVODE	Libraries	`libsundials_cvode.LIB`, `libsundials_fcvoce.a`
CVODE	Header files	`cvode/cvode.h`, `cvode/cvode_band.h`, `cvode/cvode_bandpre.h`, `cvode/cvode_bbdpre.h`, `cvode/cvode_dense.h`, `cvode/cvode_diag.h`, `cvode/cvode_direct.h`, `cvode/cvode_impl.h`, `cvode/cvode_lapack.h`, `cvode/cvode_spbcgs.h`, `cvode/cvode_spgmr.h`, `cvode/cvode_spils.h`, `cvode/cvode_sptfqmr.h`
CVODES	Libraries	`libsundials_cvodes.LIB`
CVODES	Header files	`cvodes/cvodes.h`, `cvodes/cvodes_band.h`, `cvodes/cvodes_bandpre.h`, `cvodes/cvodes_bbdpre.h`, `cvodes/cvodes_direct.h`, `cvodes/cvodes_diag.h`, `cvodes/cvodes_dense.h`, `cvodes/cvodes_impl.h`, `cvodes/cvodes_lapack.h`, `cvodes/cvodes_spbcgs.h`, `cvodes/cvodes_spgmr.h`, `cvodes/cvodes_spils.h`, `cvodes/cvodes_sptfqmr.h`
IDA	Libraries	`libsundials_ida.LIB`, `libsundials_fida.a`
IDA	Header files	`ida/ida.h`, `ida/ida_band.h`, `ida/ida_bbdpre.h`, `ida/ida_direct.h`, `ida/ida_dense.h`, `ida/ida_impl.h`, `ida/ida_lapack.h`, `ida/ida_spbcgs.h`, `ida/ida_spils.h`, `ida/ida_spgmr.h`, `ida/ida_sptfqmr.h`
IDAS	Libraries	`libsundials_idas.LIB`
IDAS	Header files	`idas/idas.h`, `idas/idas_band.h`, `idas/idas_bbdpre.h` `idas/idas_dense.h`, `idas/idas_direct.h`, `idas/idas_impl.h`, `idas/idas_lapack.h`, `idas/idas_spbcgs.h`, `idas/idas_spgmr.h`, `idas/idas_spils.h`, `idas/idas_sptfqmr.h`
KINSOL	Li-	`libsundials_kinsol.LIB`, `libsundials_fkinsol.a`

CHAPTER

NINE

APPENDIX: ARKODE CONSTANTS

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

9.1 ARKode input constants

9.1.1 ARKode main solver module

ARK_NORMAL (1): Solver returns at a specified output time.

ARK_ONE_STEP (2): Solver returns after each successful step.

9.1.2 Iterative linear solver module

PREC_NONE (0): No preconditioning.

PREC_LEFT (1): Preconditioning on the left only.

PREC_RIGHT (2): Preconditioning on the right only.

PREC_BOTH (3): Preconditioning on both the left and the right.

MODIFIED_GS (1): Use modified Gram-Schmidt procedure.

CLASSICAL_GS (2): Use classical Gram-Schmidt procedure.

9.2 ARKode output constants

9.2.1 ARKode main solver module

ARK_SUCCESS (0): Successful function return.

ARK_TSTOP_RETURN (1): ARKode succeeded by reachign the specified stopping point.

ARK_ROOT_RETURN (2): ARKode succeeded and found one more more roots.

ARK_WARNING (99): ARKode succeeded but an unusual situation occurred.

ARK_TOO_MUCH_WORK (-1): The solver took `mxstep` internal steps but could not reach `tout`.

ARK_TOO_MUCH_ACC (-2): The solver could not satisfy the accuracy demanded by the user for some internal step.

ARK_ERR_FAILURE (-3): Error test failures occurred too many times during one internal time step, or the minimum step size was reached.

ARK_CONV_FAILURE (-4): Convergence test failures occurred too many times during one internal time step, or the minimum step size was reached.

ARK_LINIT_FAIL (-5): The linear solver's initialization function failed.

ARK_LSETUP_FAIL (-6): The linear solver's setup function failed in an unrecoverable manner.

ARK_LSOLVE_FAIL (-7): The linear solver's solve function failed in an unrecoverable manner.

ARK_RHSFUNC_FAIL (-8): The right-hand side function failed in an unrecoverable manner.

ARK_FIRST_RHSFUNC_ERR (-9): The right-hand side function failed at the first call.

ARK_REPTD_RHSFUNC_ERR (-10): The right-hand side function had repeated recoverable errors.

ARK_UNREC_RHSFUNC_ERR (-11): The right-hand side function had a recoverable error, but no recovery is possible.

ARK_RTFUNC_FAIL (-12): The rootfinding function failed in an unrecoverable manner.

ARK_LFREE_FAIL (-13): The linear solver's memory deallocation function failed.

ARK_MASSINIT_FAIL (-14): The mass matrix linear solver's initialization function failed.

ARK_MASSSETUP_FAIL (-15): The mass matrix linear solver's setup function failed in an unrecoverable manner.

ARK_MASSSOLVE_FAIL (-16): The mass matrix linear solver's solve function failed in an unrecoverable manner.

ARK_MASSFREE_FAIL (-17): The mass matrix linear solver's memory deallocation function failed.

ARK_MASSMULT_FAIL (-17): The mass matrix-vector product function failed.

ARK_MEM_FAIL (-20): A memory allocation failed.

ARK_MEM_NULL (-21): The `arkode_mem` argument was `NULL`.

ARK_ILL_INPUT (-22): One of the function inputs is illegal.

ARK_NO_MALLOC (-23): The ARKode memory block was not allocated by a call to `ARKodeMalloc()`.

ARK_BAD_K (-24): The derivative order k is larger than allowed.

ARK_BAD_T (-25): The time t is outside the last step taken.

ARK_BAD_DKY (-26): The output derivative vector is `NULL`.

ARK_TOO_CLOSE (-27): The output and initial times are too close to each other.

9.2.2 ARKDLS linear solver modules

ARKDLS_SUCCESS (0): Successful function return.

ARKDLS_MEM_NULL (-1): The `arkode_mem` argument was `NULL`.

ARKDLS_LMEM_NULL (-2): The ARKDLS linear solver has not been initialized.

AKRDLS_ILL_INPUT (-3): The ARKDLS solver is not compatible with the current NVECTOR module.

ARKDLS_MEM_FAIL (-4): A memory allocation request failed.

ARKDLS_MASSMEM_FAIL (-5): A memory allocation request failed for the mass matrix solver.

ARKDLS_JACFUNC_UNRECVR (-6): The Jacobian function failed in an unrecoverable manner.

ARKDLS_JACFUNC_RECVR (-7): The Jacobian function had a recoverable error.

ARKDLS_MASSFUNC_UNRECVR (-8): The mass matrix function failed in an unrecoverable manner.

ARKDLS_MASSFUNC_RECVR (-9): The mass matrix function had a recoverable error.

9.2.3 ARKSPILS linear solver modules

ARKSPILS_SUCCESS (0): Successful function return.

ARKSPILS_MEM_NULL (-1): The `arkode_mem` argument was `NULL`.

ARKSPILS_LMEM_NULL (-2): The ARKSPILS linear solver has not been initialized.

AKRSPILS_ILL_INPUT (-3): The ARKSPILS solver is not compatible with the current NVECTOR module, or an input value was illegal.

ARKSPILS_MEM_FAIL (-4): A memory allocation request failed.

ARKSPILS_PMEM_FAIL (-5): The preconditioner module has not been initialized.

ARKSPILS_MASSMEM_FAIL (-6): A memory allocation request failed in the mass matrix solver.

9.2.4 ARKSPGMR generic linear solver module

SPGMR_SUCCESS (0): Converged.

SPGMR_RES_REDUCED (1): No convergence, but the residual norm was reduced.

SPGMR_CONV_FAIL (2): Failure to converge.

SPGMR_QRFACT_FAIL (3): A singular matrix was found during the QR factorization.

SPGMR_PSOLVE_FAIL_REC (4): The preconditioner solve function failed recoverably.

SPGMR_ATIMES_FAIL_REC (5): The Jacobian-times-vector function failed recoverably.

SPGMR_PSET_FAIL_REC (6): The preconditioner setup function failed recoverably.

SPGMR_MEM_NULL (-1): The SPGMR memory is `NULL`

SPGMR_ATIMES_FAIL_UNREC (-2): The Jacobian-times-vector function failed unrecoverably.

SPGMR_PSOLVE_FAIL_UNREC (-3): The preconditioner solve function failed unrecoverably.

SPGMR_GS_FAIL (-4): Failure in the Gram-Schmidt procedure.

SPGMR_QRSOL_FAIL (-5): The matrix R was found to be singular during the QR solve phase.

SPGMR_PSET_FAIL_UNREC (-6): The preconditioner setup function failed unrecoverably.

9.2.5 ARKSPBCG generic linear solver module

SPBCG_SUCCESS (0): Converged.

SPBCG_RES_REDUCED (1): No convergence, but the residual norm was reduced.

SPBCG_CONV_FAIL (2): Failure to converge.

SPBCG_PSOLVE_FAIL_REC (3): The preconditioner solve function failed recoverably.

SPBCG_ATIMES_FAIL_REC (4): The Jacobian-times-vector function failed recoverably.

SPBCG_PSET_FAIL_REC (5): The preconditioner setup function failed recoverably.

SPBCG_MEM_NULL (-1): The SPBCG memory is `NULL`

SPBCG_ATIMES_FAIL_UNREC (-2): The Jacobian-times-vector function failed unrecoverably.

SPBCG_PSOLVE_FAIL_UNREC (-3): The preconditioner solve function failed unrecoverably.

SPBCG_PSET_FAIL_UNREC (-4): The preconditioner setup function failed unrecoverably.

9.2.6 ARKSPTFQMR generic linear solver module

SPTFQMR_SUCCESS (0): Converged.

SPTFQMR_RES_REDUCED (1): No convergence, but the residual norm was reduced.

SPTFQMR_CONV_FAIL (2): Failure to converge.

SPTFQMR_PSOLVE_FAIL_REC (3): The preconditioner solve function failed recoverably.

SPTFQMR_ATIMES_FAIL_REC (4): The Jacobian-times-vector function failed recoverably.

SPTFQMR_PSET_FAIL_REC (5): The preconditioner setup function failed recoverably.

SPTFQMR_MEM_NULL (-1): The SPTFQMR memory is `NULL`

SPTFQMR_ATIMES_FAIL_UNREC (-2): The Jacobian-times-vector function failed.

SPTFQMR_PSOLVE_FAIL_UNREC (-3): The preconditioner solve function failed unrecoverably.

SPTFQMR_PSET_FAIL_UNREC (-4): The preconditioner setup function failed unrecoverably.

9.2.7 ARKSPFGMR generic linear solver module

SPFGMR_SUCCESS (0): Converged.

SPFGMR_RES_REDUCED (1): No convergence, but the residual norm was reduced.

SPFGMR_CONV_FAIL (2): Failure to converge.

SPFGMR_QRFACT_FAIL (3): A singular matrix was found during the QR factorization.

SPFGMR_PSOLVE_FAIL_REC (4): The preconditioner solve function failed recoverably.

SPFGMR_ATIMES_FAIL_REC (5): The Jacobian-times-vector function failed recoverably.

SPFGMR_PSET_FAIL_REC (6): The preconditioner setup function failed recoverably.

SPFGMR_MEM_NULL (-1): The SPFGMR memory is `NULL`

SPFGMR_ATIMES_FAIL_UNREC (-2): The Jacobian-times-vector function failed unrecoverably.

SPFGMR_PSOLVE_FAIL_UNREC (-3): The preconditioner solve function failed unrecoverably.

SPFGMR_GS_FAIL (-4): Failure in the Gram-Schmidt procedure.

SPFGMR_QRSOL_FAIL (-5): The matrix R was found to be singular during the QR solve phase.

SPFGMR_PSET_FAIL_UNREC (-6): The preconditioner setup function failed unrecoverably.

9.2.8 ARKPCG generic linear solver module

PCG_SUCCESS (0): Converged.

PCG_RES_REDUCED (1): No convergence, but the residual norm was reduced.

PCG_CONV_FAIL (2): Failure to converge.

PCG_PSOLVE_FAIL_REC (3): The preconditioner solve function failed recoverably.

PCG_ATIMES_FAIL_REC (4): The Jacobian-times-vector function failed recoverably.

PCG_PSET_FAIL_REC (5): The preconditioner setup function failed recoverably.

PCG_MEM_NULL (-1): The PCG memory is `NULL`

PCG_ATIMES_FAIL_UNREC (-2): The Jacobian-times-vector function failed unrecoverably.

PCG_PSOLVE_FAIL_UNREC (-3): The preconditioner solve function failed unrecoverably.

PCG_PSET_FAIL_UNREC (-4): The preconditioner setup function failed unrecoverably.

CHAPTER

TEN

APPENDIX: BUTCHER TABLES

Here we catalog the full set of Butcher tables included in ARKode. We group these into three categories: *explicit*, *implicit* and *additive*. However, since the methods that comprise an additive Runge Kutta method are themselves explicit and implicit, their component Butcher tables are listed within their separate sections, but are referenced together in the additive section.

In each of the following tables, we use the following notation (shown for a 3-stage method):

$$\begin{array}{r|ccc} c_1 & a_{1,1} & a_{1,2} & a_{1,3} \\ c_2 & a_{2,1} & a_{2,2} & a_{2,3} \\ c_3 & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline q & b_1 & b_2 & b_3 \\ p & \tilde{b}_1 & \tilde{b}_2 & \tilde{b}_3 \end{array}$$

where here the method and embedding share stage A and c values, but use their stages z_i differently through the coefficients b and $\tilde{b}$ to generate methods of orders q (the main method) and p (the embedding, typically $q = p + 1$).

Method authors often use different naming conventions to categorize their methods. For each of the methods below, we follow a uniform naming convention:

```
NAME-S-P-Q
```

where here

- `NAME` is the author (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.

Additionally, for each method we provide a plot of the linear stability region in the complex plane. These have been computed via the following approach. For any Runge Kutta method as defined above, we may define the stability function

$$R(\eta) = 1 + \eta b[I - \eta A]^{-1} e,$$

where $e \in \mathbb{R}^s$ is a column vector of all ones, $\eta = h\lambda$ and h is the time step size. If the stability function satisfies $|R(\eta)| \leq 1$ for all eigenvalues, λ, of $\frac{\partial}{\partial y} f(t, y)$ for a given IVP, then the method will be linearly stable for that problem and step size. The stability region

$$S = \{\eta \in \mathbb{C} \ : \ |R(\eta)| \leq 1\}$$

is typically given by an enclosed region of the complex plane, so it is standard to search for the border of that region in order to understand the method. Since all complex numbers with unit magnitude may be written as $e^{i\theta}$ for some value of θ, we perform the following algorithm to trace out this boundary.

1. Define an array of values `Theta`. Since we wish for a smooth curve, and since we wish to trace out the entire boundary, we choose 10,000 linearly-spaced points from 0 to 16π. Since some angles will correspond to multiple locations on the stability boundary, by going beyond 2π we ensure that all boundary locations are plotted, and by using such a fine discretization the Newton method (next step) is more likely to converge to the root closest to the previous boundary point, ensuring a smooth plot.
2. For each value $\theta \in$ `Theta`, we solve the nonlinear equation

$$0 = f(\eta) = R(\eta) - e^{i\theta}$$

   using a finite-difference Newton iteration, using tolerance 10^{-7}, and differencing parameter $\sqrt{\varepsilon}$ ($\approx 10^{-8}$).

   In this iteration, we use as initial guess the solution from the previous value of θ, starting with an initial-initial guess of $\eta = 0$ for $\theta = 0$.
3. We then plot the resulting η values that trace the stability region boundary.

We note that for any stable IVP method, the value $\eta_0 = -\varepsilon + 0i$ is always within the stability region. So in each of the following pictures, the interior of the stability region is the connected region that includes η_0. Resultingly, methods whose linear stability boundary is located entirely in the right half-plane indicate an *A-stable* method.

10.1 Explicit Butcher tables

In the category of explicit Runge-Kutta methods, ARKode includes methods that have orders 2 through 6, with embeddings that are of orders 1 through 5.

10.1.1 Heun-Euler-2-1-2

Butcher table number 0 for `ARKodeSetERKTableNum()`. This is the default 2nd order explicit method.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline 2 & 1/2 & 1/2 \\ 1 & 1 & 0 \end{array}$$

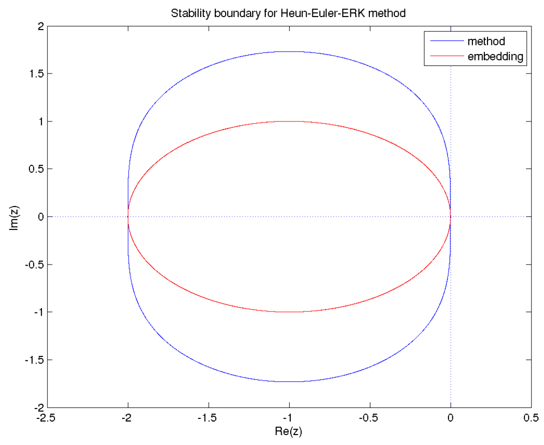


Figure 10.1: Linear stability region for the Heun-Euler method. The method's region is outlined in blue; the embedding's region is in red.

10.1.2 Bogacki-Shampine-4-2-3

Butcher table number 1 for `ARKodeSetERKTableNum()`. This is the default 3rd order explicit method.

$$
\begin{array}{r|cccc}
0 & 0 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 \\
3/4 & 0 & 3/4 & 0 & 0 \\
1 & 2/9 & 1/3 & 4/9 & 0 \\
\hline
3 & 2/9 & 1/3 & 4/9 & \\
2 & 7/24 & 1/4 & 1/3 & 1/8
\end{array}
$$

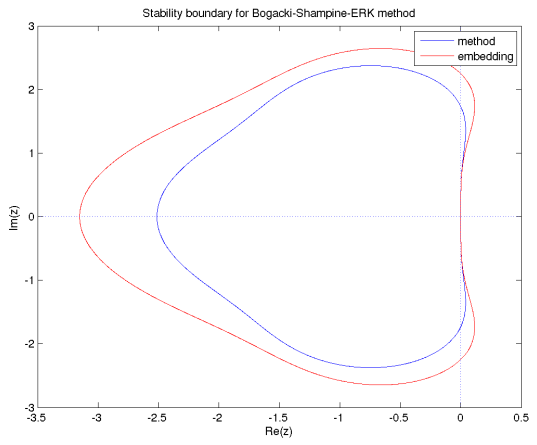


Figure 10.2: Linear stability region for the Bogacki-Shampine method. The method's region is outlined in blue; the embedding's region is in red.

10.1.3 ARK-4-2-3 (explicit)

Butcher table number 2 for `ARKodeSetERKTableNum()`. This is the explicit portion of the default 3rd order additive method.

$$
\begin{array}{r|cccc}
0 & 0 & 0 & 0 & 0 \\
\frac{1767732205903}{2027836641118} & \frac{1767732205903}{2027836641118} & 0 & 0 & 0 \\
3/5 & \frac{5535828885825}{10492691773637} & \frac{788022342437}{10882634858940} & 0 & 0 \\
1 & \frac{6485989280629}{16251701735622} & -\frac{4246266847089}{9704473918619} & \frac{10755448449292}{10357097424841} & 0 \\
\hline
3 & \frac{1471266399579}{7840856788654} & -\frac{4482444167858}{7529755066697} & \frac{11266239266428}{11593286722821} & \frac{1767732205903}{4055673282236} \\
2 & \frac{2756255671327}{12835298489170} & -\frac{10771552573575}{22201958757719} & \frac{9247589265047}{10645013368117} & \frac{2193209047091}{5459859503100}
\end{array}
$$

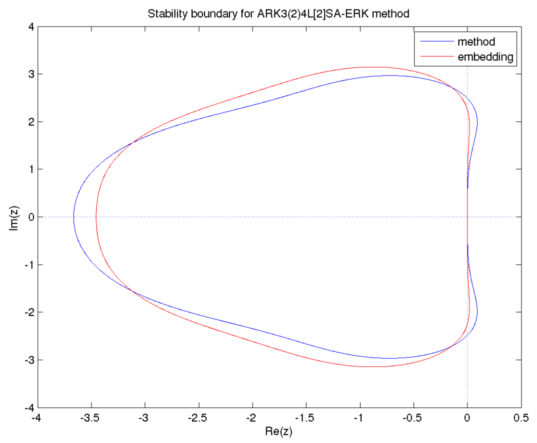


Figure 10.3: Linear stability region for the explicit ARK-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

10.1.4 Zonneveld-5-3-4

Butcher table number 3 for `ARKodeSetERKTableNum()`. This is the default 4th order explicit method.

$$
\begin{array}{r|rrrrr}
0 & 0 & 0 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 & 0 \\
1/2 & 0 & 1/2 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
3/4 & 5/32 & 7/32 & 13/32 & -1/32 & 0 \\
\hline
4 & 1/6 & 1/3 & 1/3 & 1/6 & 0 \\
3 & -1/2 & 7/3 & 7/3 & 13/6 & -16/3
\end{array}
$$

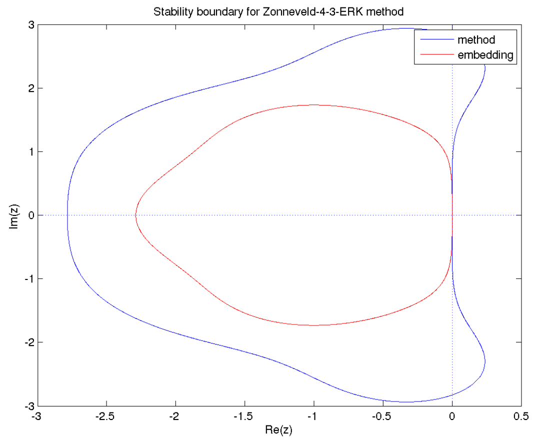


Figure 10.4: Linear stability region for the Zonneveld method. The method's region is outlined in blue; the embedding's region is in red.

10.1.5 ARK-6-3-4 (explicit)

Butcher table number 4 for `ARKodeSetERKTableNum()`. This is the explicit portion of the default 4th order additive method.

$$\begin{array}{r|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\
\frac{83}{250} & \frac{13861}{62500} & \frac{6889}{62500} & 0 & 0 & 0 & 0 \\
\frac{31}{50} & -\frac{116923316275}{2393684061468} & -\frac{2731218467317}{15368042101831} & \frac{9408046702089}{11113171139209} & 0 & 0 & 0 \\
\frac{17}{20} & -\frac{451086348788}{2902428689909} & -\frac{2682348792572}{7519795681897} & \frac{12662868775082}{11960479115383} & \frac{3355817975965}{11060851509271} & 0 & 0 \\
1 & \frac{647845179188}{3216320057751} & \frac{73281519250}{8382639484533} & \frac{552539513391}{3454668386233} & \frac{3354512671639}{8306763924573} & \frac{4040}{17871} & 0 \\
\hline
4 & \frac{82889}{524892} & 0 & \frac{15625}{83664} & \frac{69875}{102672} & -\frac{2260}{8211} & \frac{1}{4} \\
3 & \frac{4586570599}{29645900160} & 0 & \frac{178811875}{945068544} & \frac{814220225}{1159782912} & -\frac{3700637}{11593932} & \frac{61727}{225920}
\end{array}$$

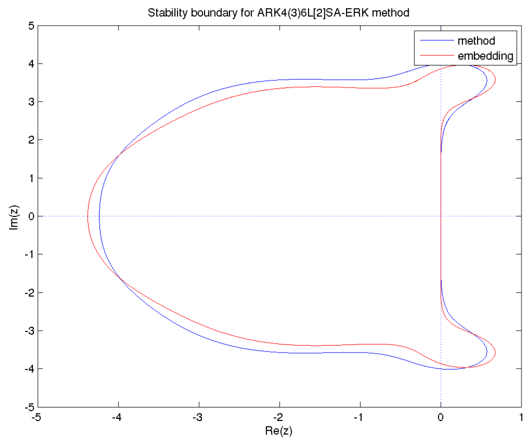


Figure 10.5: Linear stability region for the explicit ARK-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

10.1.6 Sayfy-Aburub-6-3-4

Butcher table number 5 for `ARKodeSetERKTableNum()`.

$$\begin{array}{r|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 2 & 0 & 0 & 0 & 0 \\
1 & 1/6 & 2/3 & 1/6 & 0 & 0 & 0 \\
1/2 & 0.137 & 0.226 & 0.173 & 0 & 0 & 0 \\
1 & 0.452 & -0.904 & -0.548 & 0 & 2 & 0 \\
\hline
4 & 1/6 & 1/3 & 1/12 & 0 & 1/3 & 1/12 \\
3 & 1/6 & 2/3 & 1/6 & 0 & 0 & 0
\end{array}$$

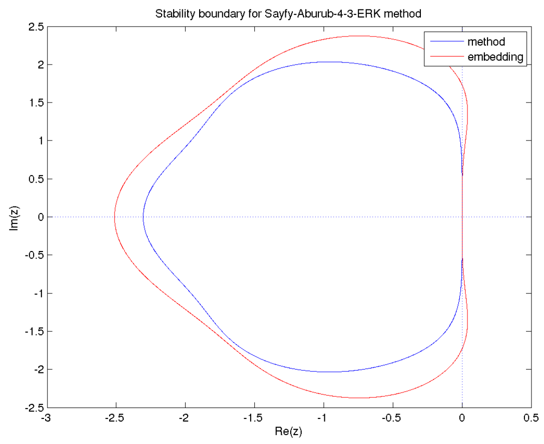


Figure 10.6: Linear stability region for the Sayfy-Aburub-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

10.1.7 Cash-Karp-6-4-5

Butcher table number 6 for `ARKodeSetERKTableNum()`. This is the default 5th order explicit method.

0	0	0	0	0	0	0
$1/5$	$1/5$	0	0	0	0	0
$3/10$	$3/40$	$9/40$	0	0	0	0
$3/5$	$3/10$	$-9/10$	$6/5$	0	0	0
1	$-11/54$	$5/2$	$-70/27$	$35/27$	0	0
$7/8$	$1631/55296$	$175/512$	$575/13824$	$44275/110592$	$253/4096$	0
5	$2825/27648$	0	$18575/48384$	$13525/55296$	$277/14336$	$1/4$
4	$37/348$	0	$250/621$	$125/594$	0	$512/1771$

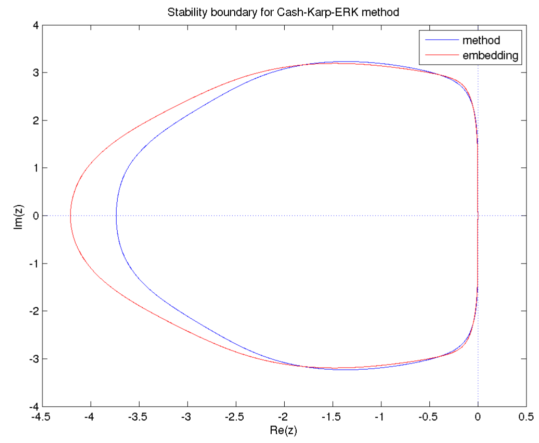


Figure 10.7: Linear stability region for the Cash-Karp method. The method's region is outlined in blue; the embedding's region is in red.

10.1.8 Fehlberg-6-4-5

Butcher table number 7 for ARKodeSetERKTableNum().

$$\begin{array}{r|ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1/4 & 1/4 & 0 & 0 & 0 & 0 & 0 \\
3/8 & 3/32 & 9/32 & 0 & 0 & 0 & 0 \\
12/13 & 1932/2197 & -7200/2197 & 7296/2197 & 0 & 0 & 0 \\
1 & 439/216 & -8 & 3680/513 & -845/4104 & 0 & 0 \\
1/2 & -8/27 & 2 & -3544/2565 & 1859/4104 & -11/40 & 0 \\
\hline
5 & 16/135 & 0 & 6656/12825 & 28561/56430 & -9/50 & 2/55 \\
4 & 25/216 & 0 & 1408/2565 & 2197/4104 & -1/5 & 0
\end{array}$$

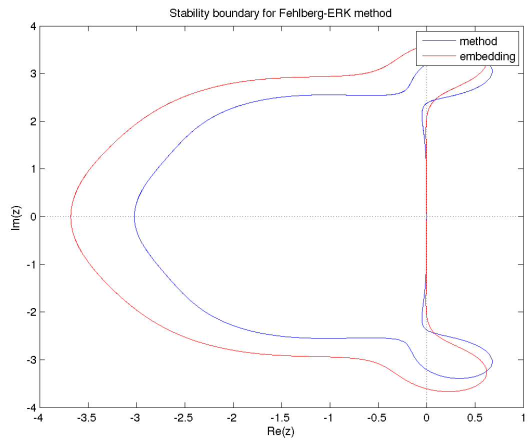


Figure 10.8: Linear stability region for the Fehlberg method. The method's region is outlined in blue; the embedding's region is in red.

10.1.9 Dormand-Prince-7-4-5

Butcher table number 8 for ARKodeSetERKTableNum().

$$\begin{array}{r|ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1/5 & 1/5 & 0 & 0 & 0 & 0 & 0 & 0 \\
3/10 & 3/40 & 9/40 & 0 & 0 & 0 & 0 & 0 \\
4/5 & 44/45 & -56/15 & 32/9 & 0 & 0 & 0 & 0 \\
8/9 & 19372/6561 & -25360/2187 & 64448/6561 & -212/729 & 0 & 0 & 0 \\
1 & 9017/3168 & -355/33 & 46732/5247 & 49/176 & -5103/18656 & 0 & 0 \\
1 & 35/384 & 0 & 500/1113 & 125/192 & -2187/6784 & 11/84 & 0 \\
\hline
5 & 35/384 & 0 & 500/1113 & 125/192 & -2187/6784 & 11/84 & 0 \\
4 & 5179/57600 & 0 & 7571/16695 & 393/640 & -92097/339200 & 187/2100 & 1/40
\end{array}$$

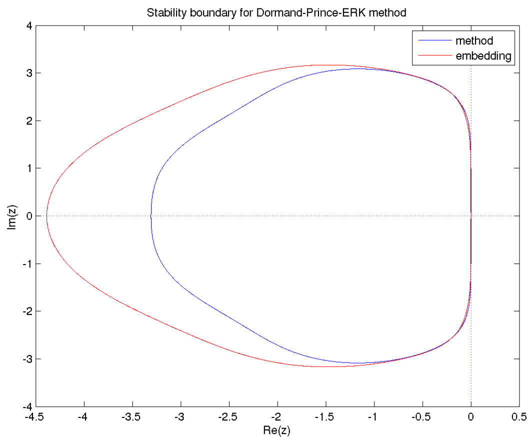


Figure 10.9: Linear stability region for the Dormand-Prince method. The method's region is outlined in blue; the embedding's region is in red.

10.1.10 ARK-8-4-5 (explicit)

Butcher table number 9 for `ARKodeSetERKTableNum()`. This is the explicit portion of the default 5th order additive method.

$$\begin{array}{r|rrrrrrrr}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{41}{100} & \frac{41}{100} & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{2935347310677}{11292855782101} & \frac{367902744464}{2072280473677} & \frac{677623207551}{8224143866563} & 0 & 0 & 0 & 0 & 0 \\
\frac{1426016391358}{7196633302097} & \frac{1268023523408}{10340822734521} & 0 & \frac{1029933939417}{13636558850479} & 0 & 0 & 0 & 0 \\
\frac{92}{100} & \frac{14463281900351}{6315353703477} & 0 & \frac{66114435211212}{5879490589093} & -\frac{54053170152839}{4284798021562} & 0 & 0 & 0 \\
\frac{24}{100} & \frac{14090043504691}{34967701212078} & 0 & \frac{15191511035443}{11219624916014} & -\frac{18461159152457}{12425892160975} & -\frac{281667163811}{9011619295870} & 0 & 0 \\
\frac{3}{5} & \frac{19230459214898}{13134317526959} & 0 & \frac{21275331358303}{2942455364971} & -\frac{38145345988419}{4862620318723} & -\frac{1}{8} & -\frac{1}{8} & 0 \\
1 & -\frac{19977161125411}{11928030595625} & 0 & -\frac{40795976796054}{6384907823539} & \frac{177454434618887}{12078138498510} & \frac{782672205425}{8267701900261} & -\frac{69563011059811}{9646580694205} & \frac{7356628}{4942186} \\
\hline
5 & -\frac{872700587467}{9133579230613} & 0 & 0 & \frac{22348218063261}{9555858737531} & -\frac{1143369518992}{8141816002931} & -\frac{39379526789629}{19018526304540} & \frac{3272738}{4290004} \\
4 & -\frac{975461918565}{9796059967033} & 0 & 0 & \frac{78070527104295}{32432590147079} & -\frac{548382580838}{3424219808633} & -\frac{33438840321285}{15594753105479} & \frac{3629800}{4656183}
\end{array}$$

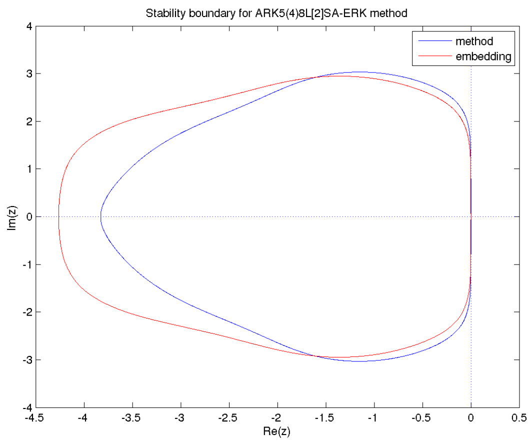


Figure 10.10: Linear stability region for the explicit ARK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

10.1.11 Verner-8-5-6

Butcher table number 10 for `ARKodeSetERKTableNum()`. This is the default 6th order explicit method.

0	0	0	0	0	0	0	0	0
1/6	1/6	0	0	0	0	0	0	0
4/15	4/75	16/75	0	0	0	0	0	0
2/3	5/6	−8/3	5/2	0	0	0	0	0
5/6	−165/64	55/6	−425/64	85/96	0	0	0	0
1	12/5	−8	4015/612	−11/36	88/255	0	0	0
1/15	−8263/15000	124/75	−643/680	−81/250	2484/10625	0	0	0
1	3501/1720	−300/43	297275/52632	−319/2322	24068/84065	0	3850/26703	0
6	3/40	0	875/2244	23/72	264/1955	0	125/11592	43/616
5	13/160	0	2375/5984	5/16	12/85	3/44	0	0

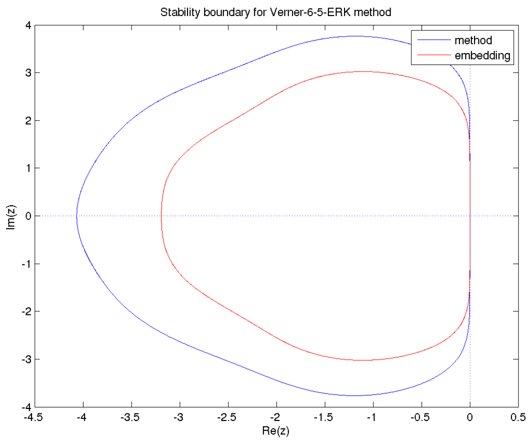


Figure 10.11: Linear stability region for the Verner-8-5-6 method. The method's region is outlined in blue; the embedding's region is in red.

10.2 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.

10.2.1 SDIRK-2-1-2

Butcher table number 11 for `ARKodeSetIRKTableNum()`. This is the default 2nd order implicit method. Both the method and embedding are A- and B-stable.

1	1	0
0	−1	1
2	1/2	1/2
1	1	0

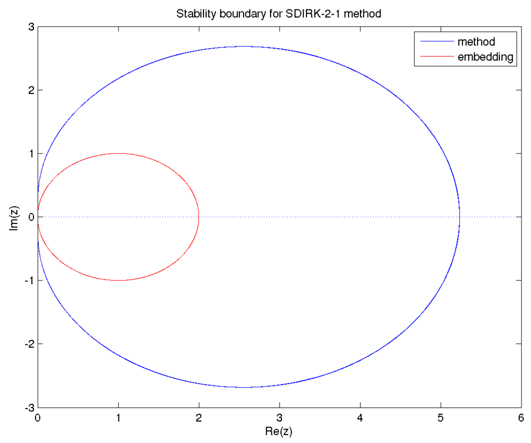


Figure 10.12: Linear stability region for the SDIRK-2-1-2 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.2 Billington-3-2-3

Butcher table number 12 for `ARKodeSetIRKTableNum()`. Here, the higher-order method is less stable than the lower-order embedding.

0.292893218813	0.292893218813	0	0
1.091883092037	0.798989873223	0.292893218813	0
1.292893218813	0.740789228841	0.259210771159	0.292893218813
3	0.691665115992	0.503597029883	-0.195262145876
2	0.740789228840	0.259210771159	0

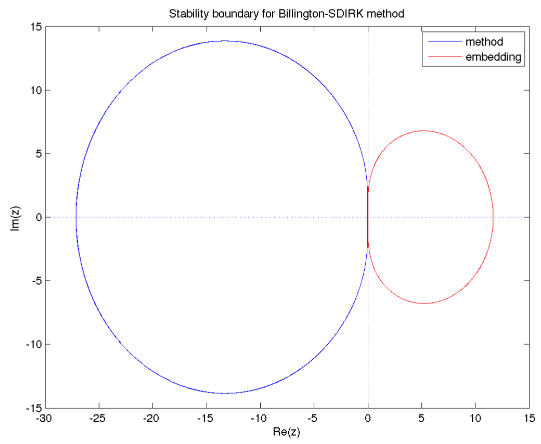


Figure 10.13: Linear stability region for the Billington method. The method's region is outlined in blue; the embedding's region is in red.

10.2.3 TRBDF2-3-2-3

Butcher table number 13 for `ARKodeSetIRKTableNum()`. As with Billington, here the higher-order method is less stable than the lower-order embedding.

$$\begin{array}{r|ccc}
0 & 0 & 0 & 0 \\
2-\sqrt{2} & \frac{2-\sqrt{2}}{2} & \frac{2-\sqrt{2}}{2} & 0 \\
1 & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{2-\sqrt{2}}{2} \\
\hline
3 & \frac{1-\frac{\sqrt{2}}{4}}{3} & \frac{\frac{3\sqrt{2}}{4}+1}{3} & \frac{2-\sqrt{2}}{6} \\
2 & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{2-\sqrt{2}}{2}
\end{array}$$

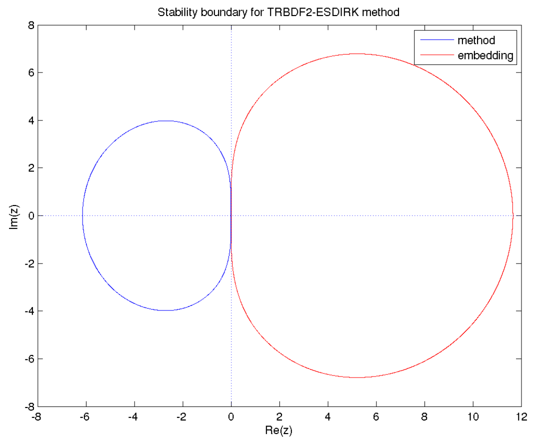


Figure 10.14: Linear stability region for the TRBDF2 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.4 Kvaerno-4-2-3

Butcher table number 14 for `ARKodeSetIRKTableNum()`. Both the method and embedding are A-stable; additionally the method is L-stable.

$$\begin{array}{r|cccc}
0 & 0 & 0 & 0 & 0 \\
0.871733043 & 0.4358665215 & 0.4358665215 & 0 & 0 \\
1 & 0.490563388419108 & 0.073570090080892 & 0.4358665215 & 0 \\
1 & 0.308809969973036 & 1.490563388254106 & -1.235239879727145 & 0.4358665215 \\
\hline
3 & 0.308809969973036 & 1.490563388254106 & -1.235239879727145 & 0.4358665215 \\
2 & 0.490563388419108 & 0.073570090080892 & 0.4358665215 & 0
\end{array}$$

10.2.5 ARK-4-2-3 (implicit)

Butcher table number 15 for `ARKodeSetIRKTableNum()`. This is the default 3rd order implicit method, and the implicit portion of the default 3rd order additive method. Both the method and embedding are A-stable; additionally

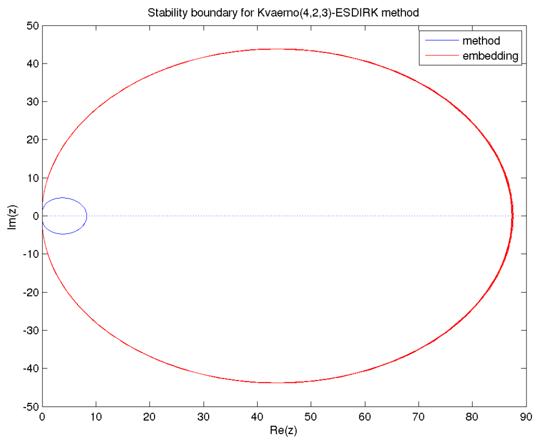


Figure 10.15: Linear stability region for the Kvaerno-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

the method is L-stable.

$$\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
\frac{1767732205903}{2027836641118} & \frac{1767732205903}{4055673282236} & \frac{1767732205903}{4055673282236} & 0 & 0 \\
\frac{3}{5} & \frac{2746238789719}{10658868560708} & -\frac{640167445237}{6845629431997} & \frac{1767732205903}{4055673282236} & 0 \\
1 & \frac{1471266399579}{7840856788654} & -\frac{4482444167858}{7529755066697} & \frac{11266239266428}{11593286722821} & \frac{1767732205903}{4055673282236} \\
\hline
3 & \frac{1471266399579}{7840856788654} & -\frac{4482444167858}{7529755066697} & \frac{11266239266428}{11593286722821} & \frac{1767732205903}{4055673282236} \\
2 & \frac{2756255671327}{12835298489170} & -\frac{10771552573575}{22201958757719} & \frac{9247589265047}{10645013368117} & \frac{2193209047091}{5459859503100}
\end{array}$$

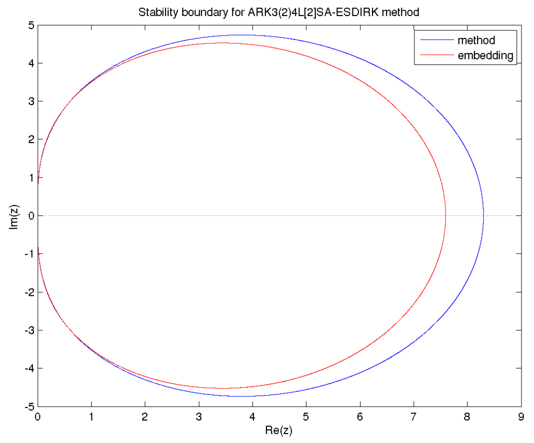


Figure 10.16: Linear stability region for the implicit ARK-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.6 Cash-5-2-4

Butcher table number 16 for `ARKodeSetIRKTableNum()`. Both the method and embedding are A-stable; additionally the method is L-stable.

$$\begin{array}{r|ccccc}
0.435866521508 & 0.435866521508 & 0 & 0 & 0 & 0 \\
-0.7 & -1.13586652150 & 0.435866521508 & 0 & 0 & 0 \\
0.8 & 1.08543330679 & -0.721299828287 & 0.435866521508 & 0 & 0 \\
0.924556761814 & 0.416349501547 & 0.190984004184 & -0.118643265417 & 0.435866521508 & 0 \\
1 & 0.896869652944 & 0.0182725272734 & -0.0845900310706 & -0.266418670647 & 0.435866521508 \\
\hline
4 & 0.896869652944 & 0.0182725272734 & -0.0845900310706 & -0.266418670647 & 0.435866521508 \\
2 & 1.05646216107052 & -0.0564621610705236 & 0 & 0 & 0
\end{array}$$

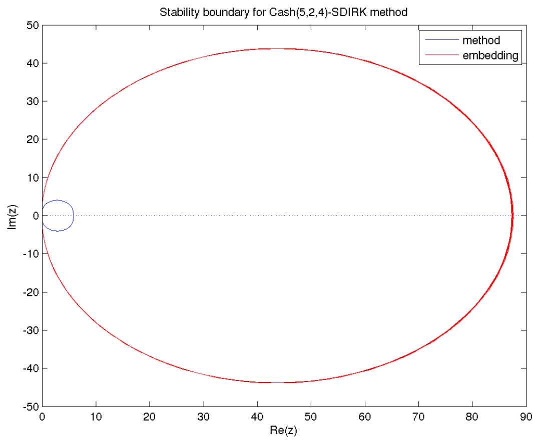


Figure 10.17: Linear stability region for the Cash-5-2-4 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.7 Cash-5-3-4

Butcher table number 17 for `ARKodeSetIRKTableNum()`. Both the method and embedding are A-stable; additionally the method is L-stable.

$$\begin{array}{r|ccccc}
0.435866521508 & 0.435866521508 & 0 & 0 & 0 & 0 \\
-0.7 & -1.13586652150 & 0.435866521508 & 0 & 0 & 0 \\
0.8 & 1.08543330679 & -0.721299828287 & 0.435866521508 & 0 & 0 \\
0.924556761814 & 0.416349501547 & 0.190984004184 & -0.118643265417 & 0.435866521508 & 0 \\
1 & 0.896869652944 & 0.0182725272734 & -0.0845900310706 & -0.266418670647 & 0.435866521508 \\
\hline
4 & 0.896869652944 & 0.0182725272734 & -0.0845900310706 & -0.266418670647 & 0.435866521508 \\
3 & 0.776691932910 & 0.0297472791484 & -0.0267440239074 & 0.220304811849 & 0
\end{array}$$

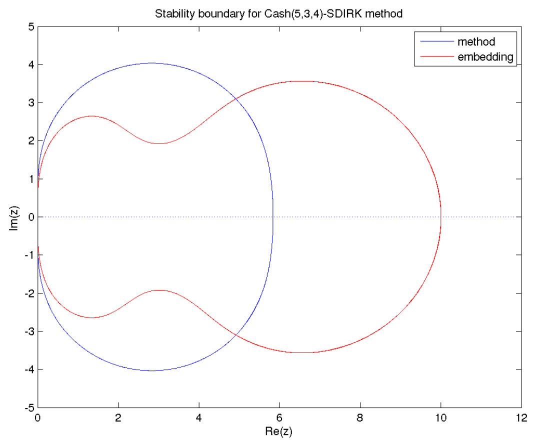


Figure 10.18: Linear stability region for the Cash-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.8 SDIRK-5-3-4

Butcher table number 18 for `ARKodeSetIRKTableNum()`. This is the default 4th order implicit method. Here, the method is both A- and L-stable, although the embedding has reduced stability.

$$\begin{array}{r|ccccc}
1/4 & 1/4 & 0 & 0 & 0 & 0 \\
3/4 & 1/2 & 1/4 & 0 & 0 & 0 \\
11/20 & 17/50 & -1/25 & 1/4 & 0 & 0 \\
1/2 & 371/1360 & -137/2720 & 15/544 & 1/4 & 0 \\
1 & 25/24 & -49/48 & 125/16 & -85/12 & 1/4 \\
\hline
4 & 25/24 & -49/48 & 125/16 & -85/12 & 1/4 \\
3 & 59/48 & -17/96 & 225/32 & -85/12 & 0
\end{array}$$

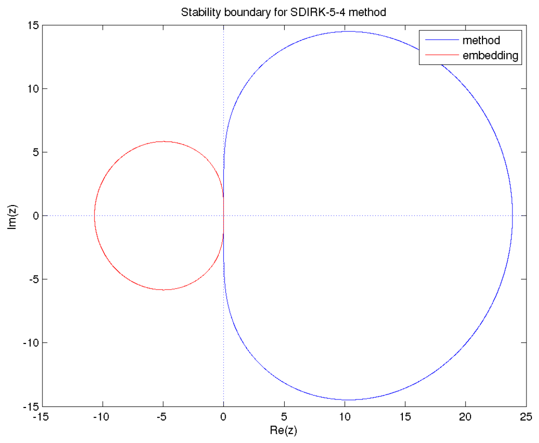


Figure 10.19: Linear stability region for the SDIRK-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.9 Kvaerno-5-3-4

Butcher table number 19 for `ARKodeSetIRKTableNum()`. Both the method and embedding are A-stable.

$$\begin{array}{r|ccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0.871733043 & 0.4358665215 & 0.4358665215 & 0 & 0 & 0 \\
0.468238744853136 & 0.140737774731968 & -0.108365551378832 & 0.4358665215 & 0 & 0 \\
1 & 0.102399400616089 & -0.376878452267324 & 0.838612530151233 & 0.4358665215 & 0 \\
1 & 0.157024897860995 & 0.117330441357768 & 0.61667803039168 & -0.326899891110444 & 0.4358665215 \\
\hline
4 & 0.157024897860995 & 0.117330441357768 & 0.61667803039168 & -0.326899891110444 & 0.4358665215 \\
3 & 0.102399400616089 & -0.376878452267324 & 0.838612530151233 & 0.4358665215 & 0
\end{array}$$

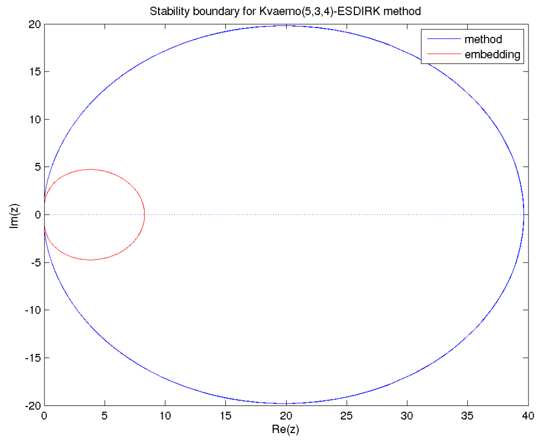


Figure 10.20: Linear stability region for the Kvaerno-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.10 ARK-6-3-4 (implicit)

Butcher table number 20 for `ARKodeSetIRKTableNum()`. This is the implicit portion of the default 4th order additive method. Both the method and embedding are A-stable; additionally the method is L-stable.

$$\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
\frac{83}{250} & \frac{8611}{62500} & -\frac{1743}{31250} & \frac{1}{4} & 0 & 0 & 0 \\
\frac{31}{50} & \frac{5012029}{34652500} & -\frac{654441}{2922500} & \frac{174375}{388108} & \frac{1}{4} & 0 & 0 \\
\frac{17}{20} & \frac{15267082809}{155376265600} & -\frac{71443401}{120774400} & \frac{730878875}{902184768} & \frac{2285395}{8070912} & \frac{1}{4} & 0 \\
1 & \frac{82889}{524892} & 0 & \frac{15625}{83664} & \frac{69875}{102672} & -\frac{2260}{8211} & \frac{1}{4} \\
\hline
4 & \frac{82889}{524892} & 0 & \frac{15625}{83664} & \frac{69875}{102672} & -\frac{2260}{8211} & \frac{1}{4} \\
3 & \frac{4586570599}{29645900160} & 0 & \frac{178811875}{945068544} & \frac{814220225}{1159782912} & -\frac{3700637}{11593932} & \frac{61727}{225920}
\end{array}$$

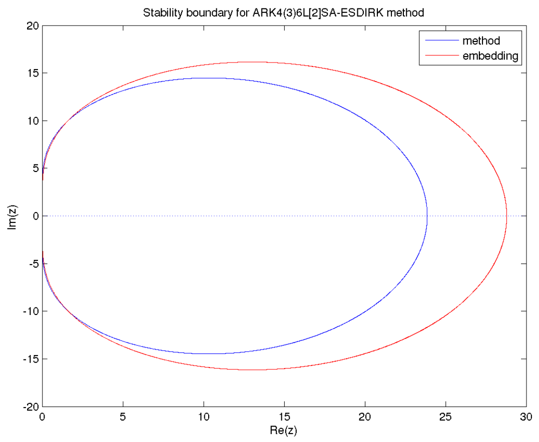


Figure 10.21: Linear stability region for the implicit ARK-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.11 Kvaerno-7-4-5

Butcher table number 21 for `ARKodeSetIRKTableNum()`. Both the method and embedding are A-stable; additionally the method is L-stable.

0	0	0	0	0	
0.52	0.26	0.26	0	0	
1.230333209967908	0.13	0.84033320996790809	0.26	0	
0.895765984350076	0.22371961478320505	0.47675532319799699	-0.06470895363112615	0.26	
0.436393609858648	0.16648564323248321	0.10450018841591720	0.03631482272098715	-0.13090704451073998	
1	0.13855640231268224	0	-0.04245337201752043	0.02446657898003141	0.61
1	0.13659751177640291	0	-0.05496908796538376	-0.04118626728321046	0.62
5	0.13659751177640291	0	-0.05496908796538376	-0.04118626728321046	0.62
4	0.13855640231268224	0	-0.04245337201752043	0.02446657898003141	0.61

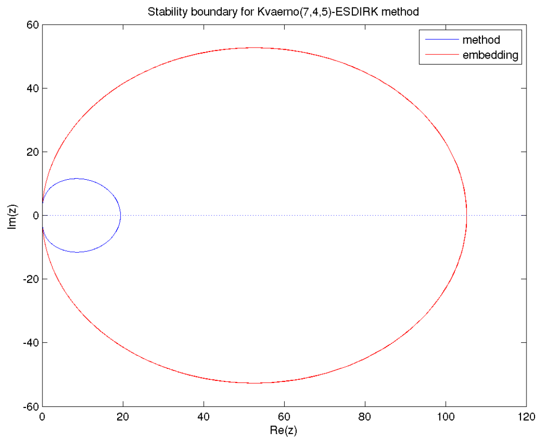


Figure 10.22: Linear stability region for the Kvaerno-7-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

10.2.12 ARK-8-4-5 (implicit)

Butcher table number 22 for `ARKodeSetIRKTableNum()`. This is the default 5th order implicit method, and the implicit portion of the default 5th order additive method. Both the method and embedding are A-stable; additionally the method is L-stable.

$$
\begin{array}{r|rrrrrrr}
0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
\frac{41}{100} & \frac{41}{200} & \frac{41}{200} & 0 & 0 & 0 & 0 & \\
\frac{2935347310677}{11292855782101} & \frac{41}{400} & -\frac{567603406766}{11931857230679} & \frac{41}{200} & 0 & 0 & 0 & \\
\frac{1426016391358}{7196633302097} & \frac{683785636431}{9252920307686} & 0 & -\frac{110385047103}{1367015193373} & \frac{41}{200} & 0 & 0 & \\
\frac{92}{100} & \frac{3016520224154}{10081342136671} & 0 & \frac{30586259806659}{12414158314087} & -\frac{22760509404356}{11113319521817} & \frac{41}{200} & 0 & \\
\frac{24}{100} & \frac{218866479029}{1489978393911} & 0 & \frac{638256894668}{5436446318841} & -\frac{1179710474555}{5321154724896} & -\frac{60928119172}{8023461067671} & \frac{41}{200} & \\
\frac{3}{5} & \frac{1020004230633}{5715676835656} & 0 & \frac{25762820946817}{25263940353407} & -\frac{2161375909145}{9755907335909} & -\frac{211217309593}{5846859502534} & -\frac{4269925059573}{7827059040749} & \\
1 & -\frac{872700587467}{9133579230613} & 0 & 0 & \frac{22348218063261}{9555858737531} & -\frac{1143369518992}{8141816002931} & -\frac{39379526789629}{19018526304540} & \frac{327273}{429000} \\
\hline
5 & -\frac{872700587467}{9133579230613} & 0 & 0 & \frac{22348218063261}{9555858737531} & -\frac{1143369518992}{8141816002931} & -\frac{39379526789629}{19018526304540} & \frac{327273}{429000} \\
4 & -\frac{975461918565}{9796059967033} & 0 & 0 & \frac{78070527104295}{32432590147079} & -\frac{548382580838}{3424219808633} & -\frac{33438840321285}{15594753105479} & \frac{36298}{465618}
\end{array}
$$

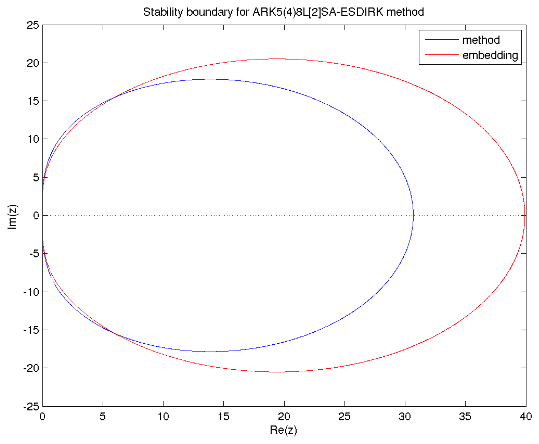


Figure 10.23: Linear stability region for the implicit ARK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

10.3 Additive Butcher tables

In the category of additive Runge-Kutta methods for split implicit and explicit calculations, ARKode includes methods that have orders 3 through 5, with embeddings that are of orders 2 through 4. These Butcher table pairs are as follows:

- 3rd-order pair: *ARK-4-2-3 (explicit)* with *ARK-4-2-3 (implicit)*, corresponding to Butcher tables 2 and 15 for `ARKodeSetARKTableNum()`.
- 4th-order pair: *ARK-6-3-4 (explicit)* with *ARK-6-3-4 (implicit)*, corresponding to Butcher tables 4 and 20 for `ARKodeSetARKTableNum()`.
- 5th-order pair: *ARK-8-4-5 (explicit)* with *ARK-8-4-5 (implicit)*, corresponding to Butcher tables 9 and 22 for `ARKodeSetARKTableNum()`.

BIBLIOGRAPHY

[BH1989] P.N. Brown and A.C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49-91, 1989.

[B1992] G.D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pp. 323-356, Oxford University Press, 1992.

[G1991] K. Gustafsson. Control theoretic techniques for stepsize selection in explicit Runge-Kutta methods. *ACM Trans. Math. Soft.*, 17:533-554, 1991.

[G1994] K. Gustafsson. Control-theoretic techniques for stepsize selection in implicit Runge-Kutta methods. *ACM Trans. Math. Soft.* 20:496-512, 1994.

[HW1993] E. Hairer, S. Norsett and G. Wanner. Solving Ordinary Differential Equations I. *Springer Series in Computational Mathematics*, vol. 8, 1993.

[HS1980] K.L. Hiebert and L.F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[HS2012] A.C. Hindmarsh and R. Serban. User Documentation for CVODE v2.7.0. Technical Report UCRL-SM-208108, LLNL, March 2012.

[HT1998] A.C. Hindmarsh and A.G. Taylor. PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-IL-129739, LLNL, February 1998.

[KC2003] C.A. Kennedy and M.H. Carpenter. Additive Runge-Kutta schemes for convection-diffusion-reaction equations. *Appl. Numer. Math.*, 44:139-181, 2003.

[R2013] D.R. Reynolds. ARKode Example Documentation. Technical Report, Southern Methodist University Center for Scientific Computation, 2013.

[S1998] G. Soderlind. The automatic control of numerical integration. *CWI Quarterly*, 11:55-74, 1998.

[S2003] G. Soderlind. Digital filters in adaptive time-stepping. *ACM Trans. Math. Soft.*, 29:1-26, 2003.

[S2006] G. Soderlind. Time-step selection algorithms: Adaptivity, control and signal processing. *Appl. Numer. Math.*, 56:488-502, 2006.

[WN2011] H.F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM J. Numer. Anal.*, 49:1715-1735, 2011.