Enabling New Flexibility in the SUNDIALS Suite of Nonlinear and Differential/Algebraic Equation Solvers

DAVID J. GARDNER, Lawrence Livermore National Laboratory, USA
DANIEL R. REYNOLDS, Southern Methodist University, USA
CAROL S. WOODWARD, Lawrence Livermore National Laboratory, USA
CODY J. BALOS, Lawrence Livermore National Laboratory, USA

In recent years, the SUite of Nonlinear and DIfferential/ALgebraic equation Solvers (SUNDIALS) has been redesigned to better enable the use of application-specific and third-party algebraic solvers and data structures. Throughout this work, we have adhered to specific guiding principles that minimized the impact to current users while providing maximum flexibility for later evolution of solvers and data structures. The redesign was done through the addition of new linear and nonlinear solvers classes, enhancements to the vector class, and the creation of modern Fortran interfaces. The vast majority of this work has been performed "behind-the-scenes," with minimal changes to the user interface and no reduction in solver capabilities or performance. These changes allow SUNDIALS users to more easily utilize external solver libraries and create highly customized solvers, enabling greater flexibility on extreme-scale, heterogeneous computational architectures.

CCS Concepts: • **Mathematics of computing → Solvers**; **Ordinary differential equations**; **Differential algebraic equations**; **Nonlinear equations**.

Additional Key Words and Phrases: Numerical software, object-oriented design, time integration, nonlinear solvers, high-performance computing

1 INTRODUCTION

The SUite of Nonlinear and DIfferential/ALgebraic equation Solvers (SUNDIALS) [41] is a collection of software packages designed to solve time-dependent and nonlinear equations on large-scale, high performance computing (HPC) systems. The packages have evolved from a multi-decade history of highly efficient solution packages for ordinary differential equations (ODEs) [13, 38, 39], differential-algebraic equations (DAEs) [61], nonlinear systems [16, 18], and sensitivity analysis enabled integrators [53, 69]. The SUNDIALS codes take advantage of the many innovations this history and the methods community have delivered in terms of adaptive time step and adaptive order integrators for ODEs and DAEs [17, 19, 36, 37, 47, 50], the combination of Krylov and Newton methods for efficient nonlinear solvers [14, 15, 60], methods and software for sensitivity analysis [21, 32, 53, 56], and effective solver controls for time integrators and nonlinear solvers [20, 40, 42–44]. Over the last two decades the SUNDIALS codes have been used in a number of applications on small to large-scale computing systems [22, 28, 29, 33–35, 52, 57, 74, 77] as well as in other mathematical libraries and frameworks [3–5, 55, 62, 68, 76, 77], garnering more than 120,000 downloads in 2021.

During this time, the science simulated as well as the software needed to carry out those simulations have grown in complexity as HPC architectures have transitioned from simplistic homogeneous systems to heterogeneous systems (e.g., CPU+GPU). With the trend toward heterogeneity, the object-oriented design of the SUNDIALS

Authors' addresses: D. J. Gardner, C. S. Woodward, and C. J. Balos, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 7000 East Ave, L-561, Livermore, CA 94550; emails: {gardner48, woodward6, balos1}@llnl.gov; D. R. Reynolds, Department of Mathematics, Southern Methodist University, P.O. Box 750156, Dallas, TX 75275; email: reynolds@smu.edu.

0098-3500/2022/0-ART0 $15.00
https://doi.org/10.1145/3539801

vector structure [41] has enabled incorporating new HPC programming models [49, 63]. However, this design did not extend to other aspects of SUNDIALS. Thus, creating interfaces to utilize new algebraic solver packages targeting large-scale heterogeneous systems would require extensive redundant code with limited extensibility. This paper describes recent enhancements following an object-oriented design to ensure adaptability to new machines and programming environments, allow users to construct architecture-aware and problem-specific data structures and solvers, ease interoperability with other libraries, and provide greater flexibility to users and for future extensions. The design principles that guided this work have been: assume as little about data layouts as possible; define interfaces that allow users to provide their own data structures and solvers with little new code; ensure SUNDIALS packages can evolve and change their offerings under these interfaces; produce highly sustainable code; and minimize the impact to current users.

The rest of this paper is organized as follows. The next section gives a brief overview of the SUNDIALS packages and is followed by a discussion of the core SUNDIALS classes. Section 4 describes changes to the vector class and new vector implementations. Sections 5 and 6 discuss new classes for linear and nonlinear solvers, respectively with accompanying demonstration programs. Section 7 overviews the modern, sustainable Fortran 2003 interfaces to SUNDIALS. Lastly, Section 8 gives a summary and next steps for SUNDIALS.

Throughout the text, the presentation of new features and enhancements is generally focused at the level of interfaces and functionality while many of the implementation-specific details of the contributions are excluded. However, references to detailed descriptions in the SUNDIALS user guides are provided for the interested reader. As the user guide for each SUNDIALS package includes shared text regarding the overall SUNDIALS infrastructure, these citations refer to version 6.2.0 of the CVODE user guide [45]. For details on updating existing codes to the latest SUNDIALS version, see the "Recent Changes" section of the package's user guide and the examples included with each release. The SUNDIALS source code, example programs, and user guides can be obtained from the SUNDIALS web page [71] or GitHub repository [72].

2 SUNDIALS OVERVIEW

This section provides a brief overview of the numerical methods implemented in the six SUNDIALS packages: CVODE, CVODES, IDA, IDAS, ARKODE, and KINSOL. Additional details on each package, with the exception of ARKODE, are available in [41]. As the native vector and matrix derived classes are written for problems in real N-space, the following overview considers problems posed in $\mathbb{R}^N$. Problems posed in other spaces (e.g., $\mathbb{C}^N$) can be solved using user-defined derived classes.

2.1 CVODE and CVODES

CVODE targets stiff and nonstiff initial value problems (IVPs) for ODEs in the explicit form

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

where t is the independent variable (e.g., time), the dependent variable is $y \in \mathbb{R}^N$, $\dot{y}$ denotes $\mathrm{d}y/\mathrm{d}t$, and $f : \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}^N$. CVODE uses variable order and variable step size implicit linear multistep methods in the fixed-leading-coefficient form [19, 47] given by

$$\sum_{i=0}^{K_1} \alpha_{n,i} y_{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} f(t_{n-i}, y_{n-i}) = 0,$$

where y_n is a computed approximation to $y(t_n)$, $h_n = t_n - t_{n-1}$ is the time step size, the values K_1 and K_2 are determined by the method type and order of accuracy, and the coefficients $\alpha_{n,i}$ and $\beta_{n,i}$ are uniquely determined by the method type and order as well as the recent step size history. Adams-Moulton methods of order 1 to 12 are provided for non-stiff problems and Backward Differentiation Formulas (BDF) of order 1 to 5 for stiff cases.

Additionally, CVODE supports projection methods [30, 70] for integrating ODE systems with constraints i.e., where the solution must satisfy $g(t, y) = 0$.

CVODES [69] is an extension of CVODE for conducting forward or adjoint sensitivity analysis of ODE IVPs with respect to the parameters $p \in \mathbb{R}^{N_p}$ i.e., problems of the form,

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

2.2 IDA and IDAS

IDA targets the more general case of DAE and implicit ODE IVPs in the form

$$F(t, y, \dot{y}) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0.$$

The state is evolved using variable order and variable step size BDF methods in fixed-leading-coefficient form [12] of order $q = 1, \ldots, 5$ given by,

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} = h_n \dot{y}_n.$$

As with CVODES and CVODE, IDAS is an extension to IDA with forward and adjoint sensitivity analysis capabilities for DAE and implicit ODE IVPs of the form

$$F(t, y, \dot{y}, p) = 0, \quad y(t_0) = y_0(p), \quad \dot{y}(t_0) = \dot{y}_0(p).$$

2.3 ARKODE

The ARKODE package targets ODE IVPs in the linearly-implicit form

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

where for any t, $M(t) : \mathbb{R}^N \to \mathbb{R}^N$ is a nonsingular linear operator. In many uses cases $M(t)$ is the identity matrix while in finite element computations it is an $N \times N$ mass matrix (or a function for its action on a vector). ARKODE supplies two classes of methods where the right-hand side function may be additively partitioned as $f(t, y) = f_1(t, y) + f_2(t, y)$.

For stiff, nonstiff, and mixed stiff/nonstiff problems, where $f_1 = f^E$ may contain the nonstiff components, which will be integrated using an explicit method, and $f_2 = f^I$ the stiff components, which will be integrated implicitly, ARKODE provides explicit, implicit, and implicit-explicit (IMEX) additive Runge-Kutta methods [6, 7, 50, 51] with the general form

$$z_i = y_{n-1} + h_n \sum_{j=1}^{i-1} A^E_{i,j} \hat{f}^E(t_{n-1} + c^E_j h_n, z_j) + h_n \sum_{j=1}^{i} A^I_{i,j} \hat{f}^I(t_{n-1} + c^I_j h_n, z_j), \quad i = 1, \ldots, s,$$

$$y_n = y_{n-1} + h_n \sum_{i=1}^{s} \left(b^E_i \hat{f}^E(t_{n-1} + c^E_j h_n, z_i) + b^I_i \hat{f}^I(t_{n-1} + c^I_j h_n, z_i) \right),$$

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

Here, $\hat{f}^E(t, y) = M(t)^{-1} f^E(t, y)$ and $\hat{f}^I(t, y) = M(t)^{-1} f^I(t, y)$ convert the IVP from linearly implicit to explicit form, z_i are internal method stages, and $\tilde{y}_n$ is an embedded solution for temporal error estimation. The number of stages s and the coefficients $A^* \in \mathbb{R}^{s \times s}$, $b^* \in \mathbb{R}^s$, $c^* \in \mathbb{R}^s$, and $\tilde{b}^* \in \mathbb{R}^s$ are given by the explicit and implicit Butcher tables that define the method and its embedding.

For multirate problems, the right-hand side may be split into "slow" components, $f_1 = f^S$, advanced with a large step H_n and "fast" components, $f_2 = f^F$, advanced with time step, $h_n \ll H_n$. For such problems, ARKODE

provides multirate infinitesimal step [65–67] and multirate infinitesimal GARK [64] methods. These methods utilize an s-stage Runge-Kutta method where the stages z_i are computed by solving (with a possibly different method) the auxiliary ODE IVP

$$\begin{aligned} \dot{v}(\theta) &= \Delta c_i f^F(T_{i-1} + \Delta c_i \theta, v(\theta)) + r(\theta), \quad \theta \in [0, H_n], \\ v(0) &= z_{i-1}, \end{aligned}$$

where $T_i = t_n + c_i H_n$, $\Delta c_i = c_i - c_{i-1}$, and $r(\theta)$ is a "forcing function" constructed from evaluations of $f^S(T_j, z_j)$, $j = 1, \ldots, i$, that propagates information from the slow time scale. The fast IVP solution is used for the new internal stage i.e., $z_i = v(H_n)$.

2.4 KINSOL

Finally the KINSOL package solves nonlinear systems in root-finding or fixed-point form,

$$F(u) = 0 \quad \text{and} \quad G(u) = u \tag{1}$$

respectively, where $u \in \mathbb{R}^N$, and the functions F and G map $\mathbb{R}^N \to \mathbb{R}^N$. For the root-finding case, KINSOL provides inexact and modified Newton methods [25, 59] which may optionally utilize a line search strategy [27]. For general fixed-point systems and systems with the special form $F(u) = Lu - \mathcal{N}(u) \Rightarrow G(u) = u - L^{-1}F(u)$, where L is a nonsingular linear operator (an $N \times N$ matrix or a function for its action on a vector) and $\mathcal{N}$ is (in general) nonlinear, KINSOL implements fixed-point and Picard iterations $u_{n+1} = G(u_n)$. The convergence of the iteration may be significantly accelerated by optionally applying Anderson's method [1, 75].

2.5 Integrator Nonlinear Systems

At each time step the implicit integration methods in CVODE(S), ARKODE, and IDA(S) must solve one or more nonlinear systems i.e.,

$$\begin{aligned} \text{CVODE:} \quad & y_n - h_n \beta_{n,0} f(t_n, y_n) - a_n = 0, \\ \text{ARKODE:} \quad & z_i - h_n A^I_{i,i} \hat{f}^I(t_{n-1} + c^I_j h_n, z_i) - a_i = 0, \text{ and} \\ \text{IDA:} \quad & F\left(t_n, y_n, h_n^{-1} \textstyle\sum_{i=0}^{q} \alpha_{n,i} y_{n-i}\right) = 0. \end{aligned}$$

where a_* is comprised of known data from prior time steps or stages. In adjoint sensitivity computations the same nonlinear systems are solved in the backward problem while forward sensitivity computations require solving an expanded nonlinear system that includes sensitivity equations. In all cases, these nonlinear systems can be viewed as generic root-finding or generic fixed-point problems and are solved using many of the same methods employed by KINSOL.

Note that in the forward sensitivity case CVODES and IDAS utilize the strategy discussed in [56] where a block diagonal approximation of the combined system Jacobian is used when solving the larger nonlinear system with Newton's method. This decoupling enables reusing the ODE/DAE system Jacobian without additional matrix factorizations or preconditioner setups. CVODES and IDAS leverage this structure to utilize the same nonlinear solvers and linear solver interfaces employed by the non-sensitivity-enabled versions of the integrators.

3 SUNDIALS STRUCTURE

To implement the above methods for integrating ODEs and DAEs or solving nonlinear systems, one must be able to perform the following actions:

(1) Compute vector operations e.g., add vectors, compute norms, etc.
(2) Solve the nonlinear systems that arise in implicit integration methods.

(3) Solve the linear systems that arise within nonlinear solvers or linearly-implicit ODEs (i.e., mass matrix linear systems).
(4) Compute matrix operations e.g., to explicitly construct the linear systems when necessary or to compute matrix-vector products.

These shared requirements across the packages lead directly to an updated SUNDIALS design, based on the existing N_Vector class and the new SUNMatrix, SUNLinearSolver, and SUNNonlinearSolver classes shown in Figure 1. These classes define abstract interfaces for performing vector and matrix operations as well as solving linear and nonlinear systems, respectively. All of the packages and the solver interfaces depend on the vector

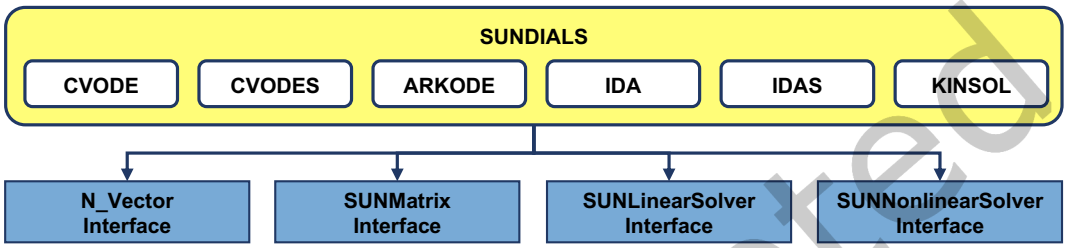

Fig. 1. An overview of the SUNDIALS packages (CVODE, CVODES, IDA, IDAS, ARKODE, and KINSOL) and the four base classes (N_Vector, SUNMatrix, SUNLinearSolver, and SUNNonlinearSolver).

interface; however, connections between the other classes depend on the particular derived class and use case e.g., matrix-free linear solvers may accept a `NULL` matrix object.

This object-oriented design encapsulates implementation-specific details behind a set of well defined abstract interfaces for performing mathematical operations. This encapsulation enables writing the SUNDIALS packages purely in terms of generic operations on objects, making them agnostic to the user's particular choice of data structures/layout, method of parallelization, algebraic solvers, etc. Moreover, the design provides flexibility, eases the addition of new features, and maximizes the reuse of shared code. With the new class interfaces discussed in this paper, SUNDIALS packages more seamlessly work with other libraries that provide complementary capabilities for scientific simulations. For example, SUNDIALS can interoperate with discretization and solver libraries as long as class implementations are provided for interfacing to other library's data structures and, if necessary, solvers.

3.1 Previous SUNDIALS Interfaces

While the vector class has long been a feature of SUNDIALS, this object-oriented approach was not originally applied to matrices or algebraic solvers. Earlier versions of SUNDIALS did clearly separate the linear solvers and matrices from the integrators and nonlinear solvers. However, these interfaces suffered from three primary challenges. First, each SUNDIALS package defined its own unique interface to each linear solver implementation, leading to significant code duplication and an unnecessary maintenance burden. Second, since each package had a separate linear solver API, it was not possible to provide a custom or third-party linear solver that could be immediately utilized by all the SUNDIALS packages. For example, a linear solver written for CVODE could not be used by ARKODE without writing additional interface code. Lastly, providing the interface functions and performing some computations required directly accessing private data within the packages, which hindered the ability of the packages to evolve without breaking user code.

Unlike the linear solvers, earlier versions of SUNDIALS did not separate the nonlinear solvers from the integrators. While the integrator-specific nonlinear systems could be written in a generic form, the nonlinear

solvers were embedded within the integrators. As such, each package implemented its own nonlinear solvers and did not provide a means for interfacing with external or user-defined nonlinear solvers leading to duplicated code and inconsistent solver options between the packages. Moreover, the lack of a shared nonlinear solver interface across the integration packages limited their flexibility and increased the software maintenance burden.

To address the challenges presented by the old linear solver interfaces and the absence of a nonlinear solver interface, we completely redesigned the handling of matrices, linear solvers, and nonlinear solvers throughout SUNDIALS. Our specific goals in this redesign were to:

- Collapse the separate package-specific linear solver interfaces into a single generic interface, thereby removing redundant code.
- Unify the package-specific nonlinear solver implementations into generic implementations shared across the integrators.
- Maintain the ability to provide integrator-specific controls over linear and nonlinear solvers for performance optimization.
- Provide streamlined interfaces with clearly defined base classes (and examples) so users can easily leverage application-specific or external algebraic solvers.

3.2 SUNDIALS Class Implementations

All of the SUNDIALS base classes follow the same fundamental design and are defined as a C structure containing a `void` pointer to the derived class member data, a virtual method table (VMT) implemented as a C structure of function pointers, and a simulation context object which provides profiling and logging capabilities. For example, the new SUNLinearSolver class is defined in `include/sundials/sundials_linearsolver.h` as follows

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;
typedef struct _generic_SUNLinearSolver_Ops *SUNLinearSolver_Ops;

struct _generic_SUNLinearSolver {
  void *content;            /* pointer to member data */
  SUNLinearSolver_Ops ops;  /* virtual method table   */
  SUNContext sunctx;        /* simulation context     */
};
```

Note the base class VMT includes pure virtual functions for which a derived class must provide an implementation, virtual functions the derived class may override, and optional functions that, if non-`NULL`, the SUNDIALS packages will utilize to improve performance. Thus, the base class pure virtual functions define the required minimum set of operations for derived classes.

Base class constructors are provided to aid in creating derived classes that are responsible for data allocations and other setup e.g., attaching method implementations to the VMT. In the linear solver case, a function to create a derived class could be implemented approximately as follows

```
SUNLinearSolver MySUNLinearSolver(...) {
  SUNLinearSolver LS = SUNLinSolNewEmpty(sunctx); /* create the base class */
  LS->content        = my_member_data;            /* set member data ptr   */
  LS->ops->solve     = my_solve_function;         /* set method pointers   */
  /* ... */
  return LS;
}
```

Additionally, base class methods are provided to copy the VMT when cloning objects thus ensuring the method pointers are set correctly. The base constructors and utility methods enable the introduction of new specialized virtual and optional methods aimed at increasing the capability and/or performance of the class with minimal impact on current users. Furthermore, when combined with the new Fortran interfaces discussed in Section 7, these methods enable the creation of custom SUNDIALS class implementations from Fortran.

4 ENHANCED VECTOR STRUCTURES

The SUNDIALS vector class has served an invaluable role in allowing existing codes to rapidly utilize SUNDIALS integrators and solvers, since users needed only to provide an N_Vector implementation to wrap and operate on application-specific data structures. Over time SUNDIALS has been progressively updated to provide a wider set of N_Vector implementations than the original serial and MPI-parallel versions, and now includes vectors based on OpenMP, Pthreads, CUDA, HIP, SYCL, and RAJA for various forms of on-node parallelism. Additionally, implementations that wrap vector objects from PETSc, *hypre*, and Trilinos have been added to streamline interfacing with other mathematical libraries. However, as hardware has advanced and become more heterogeneous, the required set of vector operations remained static, and there was not a native vector implementation that supported collecting independent vectors into a single object for partitioning data. Thus we have extended the vector class in two distinct ways:

(1) Expanded the set of vector methods to include operations that reduce the number of kernel launches, reduce memory accesses, increase arithmetic intensity, and/or minimize parallel communication.
(2) Added a "many-vector" implementation of the vector base class that enables combining different independent vector instances, each using potentially different parallelization approaches targeting distinct pieces of hardware, into a single vector.

4.1 New Vector Operations

Throughout a simulation using SUNDIALS, there exist numerous locations that use the same sets of repeated vector operations. Each of these operations over a vector of length N requires $O(N)$ floating-point operations and memory accesses. Additionally, for accelerator-based implementations, each vector operation requires a separate kernel launch, and for MPI-based parallel vectors each dot product or norm-like operation requires one `MPI_Allreduce` call. While the total floating-point work of these sequences of vector operations has remained unchanged, we have now reduced these auxiliary costs (memory accesses, kernel launches, or MPI reductions) by creating a set of "fused" operations that each combine a specific sequence of operations into a single routine.

Table 1. Subset of optional fused and vector array operations.

Function	Description
`int N_VLinearCombination(int n, realtype* c, N_Vector* X, N_Vector z)`	$z = \sum_{i=0}^{n-1} c_i X_i$
`int N_VScaleAddMulti(int n, realtype* a, N_Vector x, N_Vector* Y, N_Vector* Z)`	$Z_i = a_i\, x + Y_i$ for $i = 0, \ldots, n-1$
`int N_VDotProdMulti(int n, N_Vector x, N_Vector* Y, realtype* d)`	$d_i = x \cdot Y_i$ for $i = 0, \ldots, n-1$
`int N_VLinearSumVectorArray(int n, realtype a, N_Vector* X, realtype b, N_Vector* Y, N_Vector* Z)`	$Z_i = a\, X_i + b\, Y_i$ for $i = 0, \ldots, n-1$

We group these new operations into two categories. The first set, shown in the first three rows of Table 1, fuses standard vector operations into a single operation. The `N_VLinearCombination` and `N_VScaleAddMulti` operations replace multiple calls to `N_VLinearSum`, which computes $z = ax + by$, with a single call. Similarly, `N_VDotProdMulti` combines the reductions from several `N_VDotProd` calls, which compute $x \cdot y$, into a single reduction. The second category is composed of operations on collections of vectors that arise in sensitivity

analysis simulations. An example is shown in the last row of Table 1 where the linear sum of multiple vectors are computed in one operation.

To minimize the requirements when supplying a custom N_Vector implementation, all of these new operations are *optional* as the base class implementation calls the pre-existing sequence of *required* operations to achieve the same result. Thus, the operations can be enabled or disabled at runtime by setting the VMT function pointer to the implementation defined function or to `NULL`, thereby allowing the user to implement only the operations that are most impactful for their use case. More details on the fused operations and the full set of vector array operations that have been added to the class can be found in [45, Sections 6.1.2 and 6.1.3].

In standalone CPU-based vector tests, we have found that fused streaming operations are beneficial in cases that work on many vectors e.g., with high-order integrators and more than approximately 1,000 unknowns per MPI task. Fused reduction operations arise in orthogonalization methods in iterative methods and are beneficial for any number of vectors and with less than approximately 130K unknowns per MPI task. We note these numbers are machine dependent and will certainly change as architectures advance.

4.2 Many-vector

To enable SUNDIALS packages to leverage heterogeneous computational architectures, increase the ability for asynchronous computation, and enable greater flexibility for multiphysics applications, we have created two "many-vector" N_Vector implementations. These vectors are a software layer to enable operating on a collection of distinct vector instances as if they were a single cohesive vector. The many-vector implementations do not directly manipulate vector data; all computations are carried out by the underlying vector instances. As such, the many-vectors are designed to provide users with more flexibility in the data layout or placement on parallel and heterogeneous architectures, and, in turn, over which resources perform calculations on this data. At a high-level, the design is based on the following objectives:

- Provide a clean mechanism for users to partition their simulation data among disparate computational resources (different nodes, different compute architectures, different types of memory, etc.), enabling "hybrid" computations on heterogeneous architectures.
- Facilitate the use of a separate MPI communicator for the collection than is used for any of its subvectors, allowing the higher-level communicator to act as an MPI "intercommunicator" enabling coupled simulations between codes that utilize disjoint MPI communicator groups. From a multiphysics perspective, these disjoint groups can correspond to fully-functioning MPI-based simulations, each using its own standard MPI intracommunicator – the intercommunicator enables messages to be sent between these disjoint simulations.
- Allow computationally efficient user-supplied routines to perform operations on the relevant subvectors.
- Enable the creation of new SUNDIALS class implementations that leverage the data partitioning (e.g., partitioned integration methods, block (non)linear solvers, etc.) to improve performance for particular application types.
- Ensure seamless functionality with any existing SUNDIALS package or class.

We envision two non-mutually-exclusive scenarios where the many-vector capability will benefit users (though other use cases are possible): partitioning data between different computing resources and combining distinct MPI intracommunicators together into a multiphysics simulation.

For example, in the data partitioning case on a hybrid CPU+GPU system, a user would create subvectors using existing N_Vector constructors, some of which may be CPU-specific and others GPU-specific. In this scenario, it is assumed that all MPI-aware subvectors have been constructed using the same MPI intercommunicator. We note that some of these vectors may represent "independent" portions of the data (e.g., for processes that are not coupled between nodes); these can be constructed out of MPI-unaware N_Vectors, thereby allowing corresponding

subvector operations to avoid communication altogether. Once all subvectors have been constructed, the user then combines them together into a single many-vector by passing the subvectors to the many-vector constructor. When performing problem-specific routines (ODE RHS, DAE/nonlinear residual evaluation), the user may then access specific subvectors to retrieve and fill data, potentially copying data between accelerator and host memory as needed. An illustration of this use case for reactive flow is given in Figure 2.

The multiphysics use case corresponds to users who wish to combine separate simulations together (e.g., for fluid-structure interaction or atmosphere-ocean coupling in climate simulations) where it is assumed that MPI-aware subvectors have different MPI communicators. Here, we expect that the user will (a) create the MPI intercommunicator that connects the distinct intracommunicators together, and (b) handle communication operations between intracommunicators to couple the physical models together in their problem-defining nonlinear residual or ODE right-hand side routines. As in the data partitioning case, the user creates each of the subvectors and the intercommunicator that couples these together. The unifying many-vector is then constructed by providing the subvectors and intercommunicator to the many-vector constructor.

We further note that combinations of these two use cases are also possible via the many-vector interface. Additionally, a many-vector may be constructed using many-vectors as subvectors, thereby allowing a hierarchy of communication patterns within a simulation.

To achieve the above objectives, the MPIManyVector implementation of an N_Vector consists of a relatively lean data structure comprised primarily of an array of subvectors, the global vector length (the combined length of all subvectors for use in RMS-like norms), and an MPI communicator for global reduction operations. In support of these many-vector modules a new *required* operation, `N_VGetLength`, and a new *optional* operation, `N_VGetCommunicator`, have been added to the set of vector operations. Additionally, to minimize parallel communication with the MPIManyVector, we have expanded the set of *optional* vector operations to include local reduction kernels so that norm-like calculations may be performed with only a single `MPI_Allreduce` call (instead of one for each subvector). These methods compute only the portions of a reduction operation that are local to a given subvector and MPI rank, and their results are combined by the MPIManyVector into the overall result. As these local operations are optional, if they are not supplied by a user-provided subvector, then the MPIManyVector implementation will instead compute the final result using the original (required) vector operations (with the corresponding increase in MPI reductions).

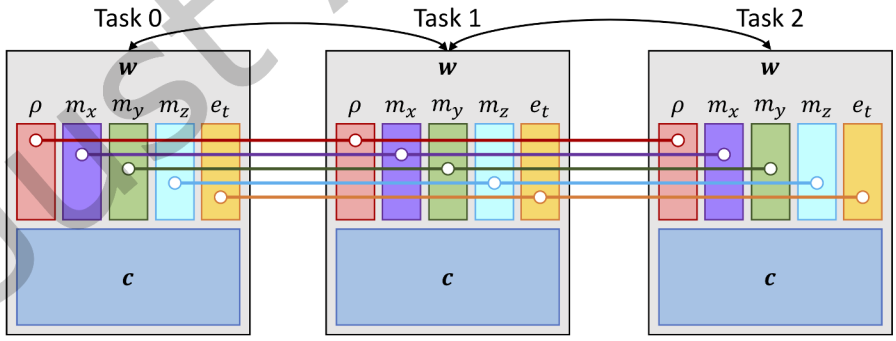

Fig. 2. Example illustration of the data partitioning many-vector use case for a reactive flow simulation. The full state, w, is composed of subvectors for the fluid variables and chemical species. The fluid quantities (density, ρ, momentum in the x, y, and z directions, m_*, and total energy, e_t) are stored as parallel vectors each connected by their own communicator. The chemical species, c, for local reactions are stored in separate task-specific vectors e.g., one or more serial, OpenMP, or GPU vectors per task.

For example consider the weighted root-mean-square (WRMS) norm of a vector x with weight vector w, both of length N, given by

$$\|x\|_{wrms} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(w_i x_i)^2}.$$

In the many-vector case, x and w above are replaced with X and W comprised of s subvectors of length N_j with the total length $N_s = \sum_{j=0}^{s-1} N_j$. The WRMS norm is now given as

$$\|X\|_{wrms} = \sqrt{\frac{1}{N_s}\sum_{j=0}^{s-1}\sum_{i=0}^{N_j-1}\left(W_{j,i}X_{j,i}\right)^2}.$$

The MPIManyVector algorithm for computing the WRMS norm is given in Algorithm 1. Here we note the use of 1 versus $s+1$ global reductions (*each* `WRMS_Norm` call requires an `MPI_Allreduce`). The full set of these optional local reduction operations can be found in [45, Section 6.1.4].

Algorithm 1: MPIManyVector weighted root-mean-square (WRMS) norm implementation

```
local_sum = 0
foreach subvector pair X_j, W_j do
    if local squared sum method exists then
        local_sum += Local_Squared_Sum(X_j, W_j)
    else
        contrib = WRMS_Norm(X_j, W_j)
        if subvector root task then
            subvec_length = Get_Length(X_j)
            local_sum += contrib * contrib * subvec_length
        end
    end
end
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, inter_comm)
return sqrt(global_sum / global_vec_length)
```

4.3 MPI+X

An added benefit of the MPIManyVector implementation is the ability for users to easily create hybrid MPI+X vectors where "X" is a type of on-node parallelism. Given an MPI-unaware on-node vector, either one provided with SUNDIALS (e.g., OpenMP, CUDA, RAJA, etc.) or one defined by the user (e.g., OpenCL, OpenACC, etc.), an MPI+X version of the vector can be simply created by supplying an MPI communicator and the on-node N_Vector to the MPIManyVector constructor. To streamline this particularly beneficial use case we have created an "MPIPlusX" vector, which is a thin wrapper of the MPIManyVector to simplify construction and provide some helpful utility functions for working with the wrapped on-node vector. See [11] for a performance comparison of various GPU on-node vectors beneath the MPIPlusX vector.

5 LINEAR SOLVERS

The new SUNLinearSolver class serves to streamline the use of problem-specific and third-party linear solvers while maintaining support for integrator-specific linear solver optimizations within SUNDIALS. To implement a custom SUNLinearSolver, a *minimum* set of required functions must be implemented. Additionally, there are a variety of optional routines that, if provided, can be leveraged by the SUNDIALS integrators to improve efficiency. There are two required SUNLinearSolver methods: `SUNLinSolGetType` and `SUNLinSolSolve` (see Table 2). The first of these allows the solver to self-identify its "type" as being either: (a) matrix-based and

Table 2. Required SUNLinearSolver operations

Function	Description
`SUNLinearSolver_Type SUNLinSolGetType(SUNLinearSolver LS)`	Returns the solver type
`int SUNLinSolSolve(SUNLinearSolver LS, SUNMatrix A, N_Vector x, N_Vector b, realtype tol)`	Solves the linear system $Ax = b$

direct, (b) matrix-based and iterative, (c) matrix-free and iterative, or (d) matrix-embedded. This identification allows the SUNDIALS packages to exploit certain solver properties for efficiency. For example, with matrix-based linear solvers the SUNDIALS integrators will infrequently update Jacobian matrices and, for direct linear solvers, matrix factorizations, to amortize the high cost of these operations across multiple time steps or nonlinear solver iterations. Similarly, for iterative linear solvers all of the SUNDIALS packages select tolerances that are "just tight enough" to aid an outer Newton iteration without performing unnecessary linear iterations. The purpose of the second required routine, `SUNLinSolSolve`, is self-explanatory – this performs the solve for a given linear system $Ax = b$.

To support integrator-specific optimizations, the SUNLinearSolver class defines a number of additional "set" routines that, if implemented, can be called to provide more fine-grained control over the SUNLinearSolver object. In addition, a set of "get" routines for retrieving artifacts (statistics, residual vectors, etc.) from the linear solver can also be called. While the SUNDIALS documentation provides an exhaustive list of these set and get routines [45, Section 8.1], we discuss a few of these optional operations (see Table 3) here to highlight their benefits in the SUNLinearSolver class.

Table 3. Subset of optional SUNLinearSolver operations

Function	Description
`int SUNLinSolSetup(SUNLinearSolver LS, SUNMatrix A)`	Performs infrequent solver setup actions
`int SUNLinSolSetATimes(SUNLinearSolver LS, void* A_data, ATimesFn ATimes)`	Provides the solver with a routine to compute the product $Ax \to z$
`int SUNLinSolSetScalingVectors(SUNLinearSolver LS, N_Vector s1, N_Vector s2)`	Provides the solver with linear system scaling vectors

The amortization of potentially high cost matrix factorizations or preconditioner setups is achieved by calling the optional `SUNLinSolSetup` function with an updated system matrix A (for matrix-based linear solvers). The

frequency of these calls depends on the type of integrator and/or nonlinear solver that uses the linear solver, and that frequency may, in many cases, be modified by the user. However, if this setup routine is not implemented by the SUNLinearSolver, then these costs must instead be incurred at every call to `SUNLinSolSolve`, which happens much more frequently. To facilitate the linear solver abstraction for matrix-based linear solvers we have created a new SUNMatrix class. This matrix class provides continued support for our pre-existing dense, banded, and sparse-direct linear solvers, and it streamlines interfacing with external linear solver packages.

All matrix-based SUNLinearSolver implementations must have a compatible SUNMatrix implementation that provides routines to clone or destroy existing SUNMatrix objects, although some SUNDIALS packages require additional matrix operations for compatibility (see Table 4). The set of SUNDIALS-provided SUNMatrix implementations, as well as their compatibility with provided N_Vector and SUNLinearSolver implementations, is discussed in detail in the SUNDIALS documentation [45, Section 7.1].

Table 4. Subset of SUNMatrix operations: (R) are required, (O) are optional

Function	Description
`SUNMatrix SUNMatClone(SUNMatrix A)`	(R) Allocates a new matrix with the same size and structure as A
`void SUNMatDestroy(SUNMatrix A)`	(R) Frees all memory associated with A
`SUNMatrix_ID SUNMatGetID(SUNMatrix A)`	(O) Identifier used by SUNLinearSolvers to check for compatibility
`int SUNMatCopy(SUNMatrix A, SUNMatrix B)`	(O) Copies all values from A into B
`int SUNMatScaleAddI(realtype c, SUNMatrix A)`	(O) Performs the combination $A \leftarrow cA + I$

For matrix-free iterative linear solvers, the SUNLinearSolver object requires knowledge of the linear system that must be solved. As such, linear solvers having this type must implement the `SUNLinSolSetATimes` routine, that is called by SUNDIALS packages to provide a function and corresponding data structure for computing the matrix-vector product, $Ax \to z$.

Lastly, for both matrix-based and matrix-free iterative solvers, the SUNLinearSolver class defines a set of routines to support both scaling and preconditioning, to balance the error between solution components, and to accelerate convergence of the linear solver. To this end, instead of solving a linear system $Ax = b$ directly, iterative SUNLinearSolver implementations may consider the transformed system $\tilde{A}\tilde{x} = \tilde{b}$, where

$$\tilde{A} = S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \qquad \tilde{b} = S_1 P_1^{-1} b, \qquad \tilde{x} = S_2 P_2 x, \tag{2}$$

and where P_1 is the left preconditioner, P_2 is the right preconditioner, S_1 is a diagonal matrix of scale factors for $P_1^{-1}b$, and S_2 is a diagonal matrix of scale factors for $P_2 x$. The SUNDIALS integrators provide these scaling matrices such that $S_1 P_1^{-1} b$ and $S_2 P_2 x$ have dimensionless components. Furthermore, SUNDIALS packages request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance:

$$\left\| \tilde{b} - \tilde{A}\tilde{x} \right\|_2 < \text{tol}.$$

These preconditioning operators and scaling matrices are provided to iterative SUNLinearSolver objects through the optional `SetPreconditioner` and `SetScalingVectors` routines. However, if an iterative linear solver implementation does not support scaling, the SUNDIALS packages will attempt to accommodate for the lack of scaling

by instead requesting that the iterative linear solver stop based on the criterion

$$\left\| P_1^{-1} b - P_1^{-1} A x \right\|_2 < \text{tol} \Big/ \bar{s}_1, \tag{3}$$

where $\bar{s}_1$ is the RMS norm of the diagonal of S_1. This tolerance adjustment is not a perfect replacement for both solution and residual scaling, in that it only accounts for the magnitude of the overall linear residual and does not balance error between specific residual or solution components.

With this SUNLinearSolver class in place, our redesign operated on two levels. On the SUNDIALS package side, we converted each linear solver interface to utilize generic linear solvers and leverage the three aforementioned types. On the solver side, we converted each existing linear solver in SUNDIALS to an implementation of the SUNLinearSolver class. We note that through the definition and use of these abstract classes alone, we were able to remove nine duplicate linear solver interfaces and three utility files in *each* of the SUNDIALS packages, while simultaneously expanding the set of linear solvers available for each package and maintaining integrator-specific efficiency enhancements. The current set of SUNDIALS-provided SUNLinearSolvers includes direct solvers for dense (native, LAPACK [2], MAGMA [73], and oneMKL [46]), banded (native and LAPACK), and sparse (KLU [23, 24], SuperLU_MT [26], SuperLU_Dist [54], and cuSOLVER [58]) matrix-based linear systems, as well as an ensemble of matrix-free scaled-preconditioned iterative linear solvers (GMRES, FGMRES, TFQMR, BiCG-Stab, and CG).

SUNLinearSolver Demonstration Problem

To demonstrate the ability for users to supply a custom linear solver for their applications, we consider the two-dimensional heat equation

$$\frac{\partial}{\partial t} u(t,x,y) = \nabla \cdot (D \nabla u(t,x,y)) + b(t,x,y).$$

Here, u is the temperature, D is the diagonal matrix of diffusion coefficients (the identity matrix in the experiments below) and the external forcing term $b(t,x,y)$ is given by

$$\begin{aligned} b(t,x,y) = &- 2\pi \sin^2(\pi x) \sin^2(\pi y) \sin(\pi t) \cos(\pi t) \\ &- D_{1,1} 2\pi^2 (\cos^2(\pi x) - \sin^2(\pi x)) \sin^2(\pi y) \cos^2(\pi t) \\ &- D_{2,2} 2\pi^2 (\cos^2(\pi y) - \sin^2(\pi y)) \sin^2(\pi x) \cos^2(\pi t). \end{aligned}$$

The problem is solved on the unit square $(x,y) \in [0,1]^2$ for $t \in [0,1]$ with $u(t,x,y) = 0$ on the boundary, and the initial condition is $u(0,x,y) = \sin^2(\pi x) \sin^2(\pi y)$. With this configuration, the problem has the analytical solution $u(t,x,y) = \sin^2(\pi x) \sin^2(\pi y) \cos^2(\pi t)$.

Spatial derivatives are discretized using second order centered finite differences with data distributed over an $n_x \times n_y$ uniform spatial grid decomposed across a two-dimensional Cartesian communicator with $p_x \times p_y$ MPI tasks. We integrate this problem in time using both CVODE and ARKODE; the ARKODE version of the problem uses the default third order diagonally implicit Runge-Kutta method in ARKODE, while the CVODE version uses adaptive order BDF methods. For both examples, the implicit systems(s) in each time step are solved using SUNDIALS' default inexact Newton method with custom SUNMatrix and SUNLinearSolver implementations that wrap a structured grid matrix and iterative linear solvers from *hypre*, CG and GMRES. The linear solvers are preconditioned with *hypre*'s PFMG parallel semicoarsening multigrid solver [8, 31]. The source files for the CVODE and ARKODE versions of this example problem are `cv_heat2D_hypre_ls.cpp` and `ark_heat2D_hypre_ls.cpp` respectively.

The member data for the custom SUNMatrix object consists of the `Hypre5ptMatrixContent` structure that holds the *hypre* structured grid, stencil, and matrix objects as well as workspace arrays and other information

Table 5. Integrator statistics from the SUNLinearSolver demonstration problem. The MF and JV columns correspond to SUNDIALS solvers with a matrix-free or a user-defined Jacobian-vector product, respectively. The statistics given are the number of successful time steps (Steps), failed time steps (Step fails), total (integrator + linear solver) ODE right-hand side evaluations (RHS evals), nonlinear iterations (NLS iters), and linear iterations (LS iters). The max error at the final time (Error) and the total average run time across 100 runs (Run time) are also given.

	SUNDIALS v6.2.0						SUNDIALS v2.7	
	CVODE			ARKODE			CVODE[a]	ARKODE[b]
CG	*hypre*	SUN MF	SUN JV	*hypre*	SUN MF	SUN JV	SUN MF	SUN MF
Steps	84	84	84	148	144	144	–	144
Step fails	4	4	4	2	1	1	–	1
RHS evals	109	609	109	1,053	5,253	1,018	–	5,250
NLS iters	106	106	106	451	435	435	–	435
LS iters	449	500	500	4,276	4,235	4,236	–	4,231
Error ($\times 10^{-7}$)	3.2	3.2	3.2	4.0	3.5	3.5	–	3.7
Run time (sec)	8.9	13.0	9.4	67.6	104.4	71.5	–	106.5
GMRES	*hypre*	SUN MF	SUN JV	*hypre*	SUN MF	SUN JV	SUN MF	SUN MF
Steps	84	80	80	147	144	144	80	144
Step fails	4	4	4	2	1	1	4	1
RHS evals	109	531	104	1,042	5,193	1,018	532	5,196
NLS iters	106	101	101	446	435	435	101	435
LS iters	394	427	427	4,014	4,175	4,174	427	4,177
Error ($\times 10^{-7}$)	3.2	11.1	11.1	3.8	3.6	3.6	11.1	3.7
Run time (sec)	8.3	12.5	9.4	68.0	110.4	79.7	12.5	109.7

[a] An interface to CG from CVODE is not available in SUNDIALS v2.7.0

[b] A patch is required to correct the norm used to test for CG convergence in SUNDIALS v2.7.0.

for updating the matrix entries. A similar structure (`HypreLSContent`) defines the member data for the SUNLinearSolver. Since the linear solver is an iterative method that uses a matrix object, the custom SUNMatrix implementation only needs to define the `GetID`, `Clone`, `Destroy`, `Copy`, and `ScaleAddI` methods from Table 4. The custom linear solver defines the required `GetType` and `Solve` methods as well as the optional `Setup` method to enable reusing the preconditioner setup. Additionally, the custom linear solver implementation provides the optional `NumIters` and `Free` methods for retrieving the number of iterations from a solve and deallocating the SUNLinearSolver and its content.

Since this custom solver does not provide a function to set scaling vectors, the integrators adjust the linear solve tolerance, as shown in (3). The main function for each example creates SUNMatrix and SUNLinearSolver instances using the custom constructors, `Hypre5ptMatrix` and `HypreLS`, respectively. Both of these constructors setup the object as outlined in Section 3. Once the matrix and linear solver objects are created, the example programs attach the objects to the integrators with the ARKODE and CVODE `SetLinearSolver` functions.

The updated linear solver interfaces make it easy to quickly compare multiple solvers and integrators on a problem. Results comparing the CG and GMRES linear solvers from SUNDIALS (matrix-free or with a user-supplied Jacobian-vector product function) and *hypre* (matrix-based), each preconditioned with PFMG, underneath a linear multistep integrator (CVODE) and a multistage integrator (ARKODE) are given in Table 5. See the files `ark_heat2D_hypre_pfmg.cpp` and `cv_heat2D_hypre_pfmg.cpp` for the versions of this example that use the SUNDIALS linear solvers. Additionally, Table 5 includes results from SUNDIALS v2.7.0[1] before the introduction of the new SUNLinearSolver class to illustrate that solver performance is unaffected by the new infrastructure (i.e., run times are within the observed run time variability of ∼2% on the machine). As only ARKODE had an interface to the SUNDIALS CG solver before the introduction of the linear solver class, results with CVODE from SUNDIALS v2.7.0 only utilize the SUNDIALS matrix-free GMRES implementation.

The tests use a 4,096 × 4,096 spatial grid distributed across 256 MPI tasks (16 tasks each in the x- and y-directions) and are run on 8 nodes of the Quartz cluster at Lawrence Livermore National Laboratory. Each compute node is composed of two Intel Xeon E5-2695 v4 chips, each with 18 cores. MPI tasks are distributed equally across the compute nodes so that 32 of the 36 cores available on a node are utilized. All integrator and solver options are set to the default values and have not been optimized for this example demonstrating the re-usability of class implementations.

In this example, as seen from Table 5, the combination of CVODE and a matrix-based linear solver from *hypre* produces the most efficient results for the desired accuracy (the integrator relative and absolute tolerances are 10^{-5} and 10^{-10} respectively). CVODE is well suited to this problem as it only requires one nonlinear solve per step, regardless of the method order, and adapts the order in addition to the time step size to maximize efficiency. ARKODE also adapts the step size, but uses a fixed order method which requires multiple implicit solves per step (for a problem better suited to the multistage IMEX methods in ARKODE see the 1D advection-reaction problem in Section 6). With the default parameters, the matrix-based *hypre* solvers lead to faster run times than the SUNDIALS solvers. The increased cost in the matrix-free case is a result of the additional right-hand side evaluations (and the corresponding MPI communication) required to approximate the Jacobian-vector product. Supplying a user-defined Jacobian-vector product function eliminates this cost. However, the SUNDIALS solvers still require approximately 0.5 more linear iterations per nonlinear iteration. The lower average number of iterations with *hypre* solvers is, in part, due to the tolerance adjustment (3) for linear solvers that do not support scaling. The performance of the SUNDIALS linear solvers can be improved for this problem by relaxing the linear solve tolerance.

Prior to the introduction of the new matrix and linear solver classes, this comparison across different integrators, solver options, and parameters would have required the user to implement unique interfaces to *hypre* solvers for both CVODE and ARKODE. With the new SUNDIALS infrastructure such interfaces can easily be reused between the integrators. Moreover, the user's calling program to setup and evolve the problem in each of these cases is essentially identical. To switch from a native SUNDIALS linear solver to using a *hypre* solver, the user only needs to swap the SUNDIALS constructor call with a call to their matrix and linear solver constructors and update the `SetLinearSolver` call to also attach a matrix object. To enable the user-defined Jacobian-vector product, only one additional function call is necessary to attach the user's function. All other linear solver related functions (e.g., adjusting the preconditioner/linear system update frequency or linear solve tolerance) are independent of the choice of SUNLinearSolver object.

[1]The implementation of CG in SUNDIALS v2.7.0 incorrectly used a WRMS norm rather than a weighted 2-norm for checking convergence. As such, the results presented are with a patched version of v2.7.0 updated to use the correct norm.

6 NONLINEAR SOLVERS

The new SUNNonlinearSolver class provides an interface for supplying SUNDIALS integrators with problem-specific and third-party nonlinear solvers. Like the other SUNDIALS base classes, this class defines a set of required and optional methods. Complete descriptions of all SUNNonlinearSolver methods can be found in the SUNDIALS documentation [45, Section 9.1].

Table 6. Required SUNNonlinearSolver operations

Function	Description
`SUNNonlinearSolver_Type SUNNonlinSolGetType(SUNNonlinearSolver NLS)`	Returns the solver type
`int SUNNonlinSolSetSysFn(SUNNonlinearSolver NLS, SUNNonlinSolSysFn SysFn)`	Set a function pointer to the nonlinear system function
`int SUNNonlinSolSolve(SUNNonlinearSolver NLS, N_Vector y0, N_Vector ycor, N_Vector w, realtype tol, booleantype callLSetup, void *mem)`	Solves the nonlinear system

The three required nonlinear solver methods are listed in Table 6. These include the `GetType` method for querying a solver object about the nonlinear system formulation it expects: root-finding, $F(u) = 0$, or fixed-point, $G(u) = 0$. Based on this information, an integrator will use the `SetSysFn` method to supply the solver with a function to evaluate the appropriate nonlinear system. As nonlinear systems in SUNDIALS integrators are formulated in terms of a correction to the predicted solution, the `Solve` method is passed the initial guess for the state vector, `y0`, as well as the correction vector, `ycor`, which contains the initial correction on input and the final correction on output. The solution is expected to satisfy the tolerance, `tol`, in the WRMS norm with weight vector, `w`. As discussed in Section 5, nonlinear solvers like Newton's method may reuse linear system information between solves to reduce the cost of setting up the linear solver. The input flag, `callLSetup`, signals whether the linear system matrix and/or preconditioner should be updated.

Rather than relying directly on the SUNLinearSolver class, the optional `SetLSetupFn` and `SetLSolveFn` methods listed in Table 7 allow the integrators to provide the nonlinear solver with functions handling integrator-specific actions for setting up and solving linear systems. These functions in turn utilize SUNLinearSolver objects. For nonlinear solvers that do not require a linear solver, or the solve is done within the nonlinear solver implementation, these optional methods do not need to be provided. Additionally, the optional `SetConvTestFn` method listed in Table 7 allows an integrator-specific convergence test function to be provided. The default stopping test for nonlinear iterations is related to the subsequent local error test for a time step, with the goal of keeping the nonlinear iteration errors from interfering with local temporal error control. In each of the integrators, this test is based on an estimate of the solver convergence rate, and incorporates several heuristics for estimating the rate of convergence and detecting solver divergence. As the default convergence test may not be optimal for all nonlinear solvers or problems, this optional method also allows a user to attach a custom convergence test that is better-suited to the solver or application. Alternately, if the nonlinear solver includes its own internal convergence test, this optional function need not be provided.

With the new SUNNonlinearSolver class in place, the integrators were updated to utilize generic nonlinear solvers, and the existing solver implementations were combined into shared derived classes. This change greatly

Table 7. Subset of optional SUNNonlinearSolver operations

Function	Description
`int SUNNonlinSolSetLSetupFn(SUNNonlinearSolver NLS, SUNNonlinSolLSetupFn SetupFn);`	Set linear solver setup function
`int SUNNonlinSolSetLSolveFn(SUNNonlinearSolver NLS, SUNNonlinSolLSolveFn SolveFn);`	Set linear solver solve function
`int SUNNonlinSolSetConvTestFn(SUNNonlinearSolver NLS, SUNNonlinSolConvTestFn CTestFn, void* ctest_data);`	Set convergence test function

improves the flexibility of the time integrators, as third party nonlinear solver interfaces (e.g., the SUNNonlinearSolver_PetscSNES class utilizing PETSc's SNES solver [9, 10]), or custom nonlinear solvers (as illustrated below) can now be created.

SUNNonlinearSolver Demonstration Problem

To demonstrate the flexibility provided by the new nonlinear solver class we consider a one-dimensional advection-reaction problem using a stiff variation of the Brusselator model from chemical kinetics [37] utilizing a custom SUNNonlinearSolver. The system is given by

$$\begin{aligned}\frac{\partial u}{\partial t} &= -c\frac{\partial u}{\partial x} + A - (w+1)u + vu^2 \\ \frac{\partial v}{\partial t} &= -c\frac{\partial v}{\partial x} + wu - vu^2 \\ \frac{\partial w}{\partial t} &= -c\frac{\partial w}{\partial x} + \frac{B-w}{\epsilon} - wu\end{aligned}$$

where u, v, and w are chemical concentrations, $t \in [0, 10]$ is time, $x \in [0, b]$ is the spatial variable, $c = 0.01$ is the advection speed, $A = 1$ and $B = 3.5$ are the concentrations of chemical species that remain constant over space and time, and $\epsilon = 5 \times 10^{-6}$ is a parameter that determines the stiffness of the system. The problem uses periodic boundary conditions and the initial condition is

$$u(0,x) = A + p(x), \qquad v(0,x) = B/A + p(x), \qquad w(0,x) = 3.0 + p(x),$$
$$p(x) = 0.1 \exp\left(-2\left(2x/b - 1\right)^2\right).$$

Spatial derivatives are discretized with first order upwind finite differences on a uniform mesh with n_x points divided across p_x MPI tasks. The state is stored in a task-local serial vector wrapped by the MPIPlusX vector. The default third order IMEX method from ARKODE is used to evolve the system in time with the advection terms treated explicitly and the reaction terms implicitly.

As the reactions are purely local in space, the implicit system that must be solved at each stage can be decomposed into independent nonlinear systems at each grid point. To exploit this locality, the user code implements a *custom problem-specific* SUNNonlinearSolver to perform MPI task-local solves with the local serial vectors. This eliminates nearly all parallel communication within the implicit solves i.e., global reductions for norms and orthogonalization in GMRES. The only global communication necessary is at the end of the solve phase to check if all the task-local nonlinear solves were successful. The source file for this example is `ark_brusselator1D_task_local_nls.c`.

Here, the custom nonlinear solver member data is defined as the `TaskLocalNewton_Content` structure containing an instance of the native SUNDIALS Newton solver and other data for solving the local nonlinear problem. The custom solver defines the required `GetType`, `SetSysFn`, and `Solve` methods as well as the optional methods: `SetConvTestFn` to use the integrator-provided convergence test function, `Free` to deallocate the solver, and `GetNumConvFails` to retrieve the number of solver failures. Additionally, the optional `Initialize` method is used, in this case, to provide the task-local instance of the SUNDIALS Newton solver with user-defined functions to evaluate the local residual (`TaskLocalNlsResidual`) and solve the local linear systems (`TaskLocalLSolve`). The main function creates a solver instance using the custom constructor, `TaskLocalNewton`, to setup the object as outlined in Section 3. Once the custom solver object is created, the example program attaches it to the integrator with `ARKStepSetNonlinearSolver`.

To compare this solution approach with what was possible before the addition of the SUNNonlinearSolver class, the example code provides an option to use the SUNDIALS Newton method to perform a global solve with GMRES as the linear solver and a user-defined, task-local preconditioner. Multiple tests are run with a fixed spatial resolution, $b/n_x = 10^{-3}$, and the domain size is increased with the number of MPI tasks; specifically we test with $b \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$ and $p_x \in \{40, 80, 160, 320, 640, 1280, 2560, 5120, 10240\}$ with 40 MPI tasks per compute node on the Lassen supercomputer at Lawrence Livermore National Laboratory. Note the results below utilize only the CPUs (two IBM Power9 processors with 44 cores total) on Lassen. For results utilizing GPU acceleration and the MPIPlusX vector with various GPU task-local vectors see [11].

Fig. 3. Speedup results for various problem sizes for the SUNNonlinearSolver demonstration problem using a custom task-local nonlinear solver (red squares) and a global Newton solve (blue circles). Execution times are the average of 20 runs and each task-local solver data point is annotated with the approximate speedup achieved compared to the global solver.

Execution times for the global and task-local solvers are given in Figure 3. Although the reduction in synchronizations does not significantly impact the problem scaling, the custom solver results in speedups of 4x to more than 7x. As such, it is clear that the new nonlinear solver class provides important new capabilities that were not possible before the flexibility enhancements and can lead to significantly more efficient solver approaches by exploiting problem structure.

7 FORTRAN 2003 INTERFACES

Fortran has historically been heavily used in scientific computing and remains widespread today. As such, although SUNDIALS is written in C, Fortran interfaces have been provided for many years. These Fortran 77 standard compliant interfaces have allowed Fortran users to access a *subset* of features from CVODE, IDA, ARKODE, and KINSOL and a *limited* number of the vector, matrix, and solver class implementations available to C/C++ users. While the Fortran 77 interfaces provided access to some SUNDIALS capabilities they also suffered from limited flexibility and sustainability. Notably, the lack of derived types in Fortran 77 was incompatible with SUNDIALS classes, and circumventing this limitation required accessing SUNDIALS classes using global memory, rendering the Fortran 77 interfaces non-threadsafe. As a result, it was difficult for Fortran users to supply application-specific data layouts or custom solvers, and it was impossible to use multiple instances of SUNDIALS packages simultaneously in a multithreaded environment. Furthermore, maintaining these interfaces required significant manual upkeep of code to "glue" the interfaces to C; thus, the Fortran 77 interfaces were not sustainable in the long-term.

To address these limitations, we have introduced new Fortran 2003 standard compliant interfaces that provide access to *all of the features in all six* of the SUNDIALS packages (access to CVODES and IDAS were previously unsupported from Fortran 77) as well as most of the core class implementations. These new interfaces are

generated automatically using the SWIG-Fortran [48] tool, which only requires minimal additional interface code, much of which is common across SUNDIALS, to create bindings using the Fortran 2003 features for interoperability with ISO C code. When the C API changes in any package or class with an existing Fortran 2003 interface, the SWIG-Fortran tool is run, and the interface "glue" code is regenerated before package release. This process fits nicely into a modern software development workflow and could be further automated to run as part of a continuous integration/continuous delivery pipeline. Thus, the Fortran 2003 interfaces have a low-maintenance cost and can quickly be extended to support new features and class implementations as they are added i.e., the Fortran 2003 interfaces are sustainable.

Since the Fortran 2003 interfaces are based on features of the Fortran 2003 standard for interopability with ISO C code, they allow Fortran users to continue to write idiomatic Fortran with few caveats, but to then interact with SUNDIALS in a way that closely resembles the C API. In fact, the Fortran 2003 interface API is close enough to the C API that the C API documentation serves dual-purpose as the Fortran 2003 documentation, modulo some information on data types and subtleties attributed to inherent differences of C and Fortran. Function signatures match the C API with an "F" prepended to the function name e.g., `FN_VConst` instead of `N_VConst`. The generic SUNDIALS classes, such as the `N_Vector`, are interfaced as Fortran derived-types of the same name. This convention means that users can easily provide custom, application-specific implementations of the generic SUNDIALS classes that are written entirely in Fortran (as illustrated below). Data arrays (e.g., real arrays) are interfaced as Fortran arrays of the corresponding type defined in the `iso_c_binding` intrinsic module. The exact mapping of SUNDIALS C types to Fortran 2003 types, as well as discussion of subtle usage differences between the C and Fortran 2003 APIs, is given in the SUNDIALS documentation [45, Section 5.1].

Fortran 2003 Interface Demonstration Problem

A second version of the demonstration problem from Section 6 utilizing a custom nonlinear solver is also implemented in Fortran. The source code for this version of the problem is `ark_brusselator1D_task_local_nls_f2003.f90`. As the Fortran interface closely mirrors the C API, the same steps to create a derived class implementation apply to the Fortran use case.

In this example, all of the data needed by the custom task-local nonlinear solver is contained in a user-defined module, `nls_mod`. As such, the data is effectively global (albeit namespace protected). An alternative, thread safe approach would be to define a new type for the solver data and attach it as the class member data. In this case, the module includes an instance of the SUNDIALS Newton method and other data needed for solving the task-local nonlinear systems as well as functions implementing required and optional nonlinear solver methods. An instance of the nonlinear solver object is created using the constructor subroutine `TaskLocalNewton` which leverages the corresponding base class constructor to create a SUNNonlinearSolver object and `c_funloc` to set the function pointers in the VMT. In this case, the object's content field does not need to be set as all the data needed by the solver is accessible through the `nls_mod` module. Once the solver and all its required data are created, the nonlinear solver object is attached to the integrator with the `FARKStepSetNonlinearSolver` function. As expected, the C and Fortran versions of the problem have the same integrator and solver statistics and nearly identical run times.

8 CONCLUSIONS

The growing complexity of scientific simulations and recent advances in high performance computing systems have resulted in increased demand for utilizing new programming models and more varied algebraic solver options with the SUNDIALS time integrators and nonlinear solvers. To meet these demands, we have created new matrix, linear solver, and nonlinear solver classes following the same object-oriented design previously utilized by the SUNDIALS vector. These new additions adhere to a number of guiding principles that provide

greater flexibility while increasing the sustainability of the SUNDIALS packages, requiring minimal changes to user codes, and resulting in significant performance benefits in many situations.

The new classes allow for easier interoperability with and reuse of external and application-specific data structures and solvers across SUNDIALS packages. In particular, the encapsulation of the nonlinear solvers from the time integrators provides flexibility in nonlinear solution approaches that was not previously available in SUNDIALS. Additionally, we have expanded the SUNDIALS vector class implementations to include a new many-vector capability, allowing for a vector comprised of various subvectors. This many-vector construct provides an easy way to accommodate multiphysics simulations as well as heterogeneous architectures where different vectors operate on data residing in potentially different memory spaces. An MPI+X vector was further added on top of the many-vector to streamline the introduction of new vectors supporting various approaches to on-node parallelism with heterogeneous computing platforms. New fused operations better utilize available hardware and enable development of reduced communication algorithms. Lastly, modern Fortran interfaces allow full access to the flexibility provided with the SUNDIALS classes.

Numerical tests have shown that the new infrastructure retains the existing behavior of the time integrators and does not increase the cost of simulations. Moreover, the results demonstrate the power of being able to easily compare the performance of fundamentally different integrators and solver approaches within the same problem. Significant speedups were also shown when leveraging the new nonlinear solver class to provide a problem-specific solver exploiting the structure and locality in the nonlinear system.

With these new capabilities in place, future work includes implementing new vectors and matrices to accommodate more types of on-node parallelism, investigating algorithms that leverage fused operations for greater performance, and expanding the set of native solver interfaces provided with SUNDIALS. Additionally, we plan to leverage the new many-vector implementation within SUNDIALS to exploit the structural features of different applications.

ACKNOWLEDGMENTS

The authors thank Alan Hindmarsh for numerous discussions and insights into details of integrator controls on solvers and design of integrator codes.

This research was supported by the Exascale Computing Project, (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Support for this work was provided in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the Frameworks, Algorithms, and Scalable Technologies for Mathematics (FASTMath) Institute.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. LLNL-JRNL-816631.

REFERENCES

[1] Donald G. Anderson. 1965. Iterative procedures for nonlinear integral equations. *J. ACM* 12, 4 (1965), 547–560. https://doi.org/10.1145/321296.321305

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/1.9780898719604

[3] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, et al. 2021. MFEM: A modular finite element methods library. *Comput. Math. with Appl.* 81 (2021), 42–74. https://doi.org/10.1016/j.camwa.2020.06.009

[4] Joel A. E. Andersson, Joris Gillis, Greg Horn, James B. Rawlings, and Moritz Diehl. 2019. CasADi – A software framework for nonlinear optimization and optimal control. *Math. Program. Comput.* 11, 1 (2019), 1–36. https://doi.org/10.1007/s12532-018-0139-4

[5] Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Thomas C. Clevenger, Marc Fehling, Alexander V. Grayver, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Reza Rastak, Ignacio Thomas, Bruno Turcksin, Zhuoran Wang, and David Wells. 2020. The deal.II Library, Version 9.2. *J. Numer. Math.* 28, 3 (2020), 131–146. https://doi.org/10.1515/jnma-2020-0043

[6] Uri M. Ascher, Steven J. Ruuth, and Raymond J. Spiteri. 1997. Implicit-Explicit Runge-Kutta Methods for Time-Dependent Partial Differential Equations. *Appl. Numer. Math.* 25, 2-3 (Nov 1997), 151–167. https://doi.org/10.1016/S0168-9274(97)00056-1

[7] Uri M. Ascher, Steven J. Ruuth, and Brian T. R. Wetton. 1995. Implicit-Explicit Methods for Time-Dependent Partial Differential Equations. *SIAM J. Numer. Anal.* 32, 3 (1995), 797–823. https://doi.org/10.1137/0732037

[8] Steven F. Ashby and Robert D. Falgout. 1996. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nucl. Sci. Eng.* 124, 1 (1996), 145–159. https://doi.org/10.13182/NSE96-A24230

[9] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2021. PETSc Web page. https://petsc.org/.

[10] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2021. *PETSc Users Manual.* Technical Report ANL-95/11 - Revision 3.15. Argonne National Laboratory.

[11] Cody J. Balos, David J. Gardner, Carol S. Woodward, and Daniel R. Reynolds. 2021. Enabling GPU Accelerated Computing in the SUNDIALS Time Integration Library. *Parallel Comput.* 108 (2021), 102836. https://doi.org/10.1016/j.parco.2021.102836

[12] Kathryn Eleda Brenan, Stephen L. Campbell, and Linda Ruth Petzold. 1995. *Numerical solution of initial-value problems in differential-algebraic equations.* SIAM Publications, Philadelphia, PA. https://doi.org/10.1137/1.9781611971224

[13] Peter N. Brown, George D. Byrne, and Alan C. Hindmarsh. 1989. VODE: A variable-coefficient ODE solver. *SIAM J. Sci. Statist. Comput.* 10, 5 (1989), 1038–1051. https://doi.org/10.1137/0910062

[14] Peter N. Brown and Alan C. Hindmarsh. 1986. Matrix-free methods for stiff systems of ODE's. *SIAM J. Numer. Anal.* 23, 3 (1986), 610–638. https://doi.org/10.1137/0723039

[15] Peter N. Brown and Alan C. Hindmarsh. 1989. Reduced storage matrix methods in stiff ODE systems. *Appl. Math. Comput.* 31 (1989), 40–91. https://doi.org/10.1016/0096-3003(89)90110-0

[16] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold. 1994. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM J. Sci. Comput.* 15, 6 (1994), 1467–1488. https://doi.org/10.1137/0915088

[17] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold. 1998. Consistent initial condition calculation for differential-algebraic systems. *SIAM J. Sci. Comput.* 19, 5 (1998), 1495–1512. https://doi.org/10.1137/S1064827595289996

[18] Peter N. Brown and Youcef Saad. 1990. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Statist. Comput.* 11, 3 (1990), 450–481. https://doi.org/10.1137/0911026

[19] George D. Byrne and Alan C. Hindmarsh. 1975. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Trans. Math. Software* 1, 1 (1975), 71–96. https://doi.org/10.1145/355626.355636

[20] George D. Byrne and Alan C. Hindmarsh. 1987. Stiff ODE solvers: A review of current and coming attractions. *J. Comput. Phys.* 70, 1 (1987), 1–62. https://doi.org/10.1016/0021-9991(87)90001-5

[21] Yang Cao, Shengtai Li, Linda Petzold, and Radu Serban. 2003. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution. *SIAM J. Sci. Comput.* 24, 3 (2003), 1076–1089. https://doi.org/10.1016/S0377-0427(02)00528-9

[22] Nicholas T. Carnevale and Michael L. Hines. 2006. *The NEURON Book.* Cambridge University Press, Cambridge, England. https://doi.org/10.1017/CBO9780511541612

[23] Tim Davis. 2020. KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[24] Tim Davis and Ekanathan Palamadai Natarajan. 2010. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Software* 37, 3 (2010), 36:1–36:17. https://doi.org/10.1145/1824801.1824814

[25] Ron S. Dembo, Stanley C. Eisenstat, and Trond Steihaug. 1982. Inexact Newton methods. *SIAM J. Numer. Anal.* 19, 2 (1982), 400–408. https://doi.org/10.1137/0719025

[26] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. 1999. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Anal. Appl.* 20, 4 (1999), 915–952. https://doi.org/10.1137/S0895479897317685

[27] John E. Dennis, Jr and Robert B. Schnabel. 1996. *Numerical methods for unconstrained optimization and nonlinear equations.* SIAM Publications, Philadelphia, PA. https://doi.org/10.1137/1.9781611971200

[28] Milo R. Dorr, J-L Fattebert, Michael E. Wickett, James F. Belak, and Patrice E.A. Turchi. 2010. A numerical algorithm for the solution of a phase-field model of polycrystalline materials. *J. Comput. Phys.* 229, 3 (2010), 626–641. https://doi.org/10.1016/j.jcp.2009.09.041

[29] B.D. Dudson, M.V. Umansky, XQ Xu, P.B. Snyder, and H.R. Wilson. 2009. BOUT++: A framework for parallel plasma fluid simulations. *Comput. Phys. Commun.* 180, 9 (2009), 1467–1480. https://doi.org/10.1016/j.cpc.2009.03.008

[30] Edda Eich. 1993. Convergence results for a coordinate projection method applied to mechanical systems with algebraic constraints. *SIAM J. Numer. Anal.* 30, 5 (1993), 1467–1482. https://doi.org/10.1137/0730076

[31] Robert D. Falgout and Jim E. Jones. 2000. Multigrid on massively parallel architectures. In *Multigrid Methods VI*. Springer, Berlin, Heidelberg, 101–107. https://doi.org/10.1007/978-3-642-58312-4_13
[32] William F. Feehery, John E. Tolsma, and Paul I. Barton. 1997. Efficient sensitivity analysis of large-scale differential-algebraic systems. *Appl. Numer. Math.* 25, 1 (1997), 41–54. https://doi.org/10.1016/S0168-9274(97)00050-0
[33] David J. Gardner, Jorge E. Guerra, François P. Hamon, Daniel R. Reynolds, Paul A. Ullrich, and Carol S. Woodward. 2018. Implicit–explicit (IMEX) Runge–Kutta methods for non-hydrostatic atmospheric models. *Geosci. Model Dev.* 11, 4 (2018), 1497–1515. https://doi.org/10.5194/gmd-11-1497-2018
[34] David J. Gardner, Carol S. Woodward, Daniel R. Reynolds, Gregg Hommes, Sylvie Aubry, and A. Arsenlis. 2015. Implicit integration methods for dislocation dynamics. *Modelling Simul. Mater. Sci. Eng.* 23, 2 (2015), 025006. https://doi.org/10.1088/0965-0393/23/2/025006
[35] David G. Goodwin, Raymond L. Speth, Harry K. Moffat, and Bryan W. Weber. 2018. Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes. https://www.cantera.org. https://doi.org/10.5281/zenodo.1174508 Version 2.4.0.
[36] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. 1993. *Solving Ordinary Differential Equations I: Nonstiff Problems* (2 ed.). Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78862-1
[37] Ernst Hairer and Gerhard Wanner. 1996. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems* (2 ed.). Springer-Verlag, Berlin Heidelberg. https://doi.org/10.1007/978-3-642-05221-7
[38] Alan C. Hindmarsh. 1980. LSODE and LSODI, two new initial value ordinary differential equation solvers. *ACM SIGNUM Newslett.* 15, 4 (1980), 10–11. https://doi.org/10.1145/1218052.1218054
[39] Alan C. Hindmarsh. 1983. ODEPACK, a systematized collection of ODE solvers. In *Scientific Computing: Applications of Mathematics and Computing to the Physical Sciences (IMACS transactions on scientific computation, Vol. 1)*, Robert S. Stepleman, M. Carver, William F. Ames, et al. (Eds.). North-Holland, Amsterdam, Netherlands, 55–64.
[40] Alan C. Hindmarsh. 1992. Detecting stability barriers in BDF solvers. In *Computational Ordinary Differential Equations (Institute of Mathematics and its Applications Conference Series)*, J. R. Cash and I. Gladwell (Eds.). Oxford University Press, Oxford, England, 87–96.
[41] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Software* 31, 3 (2005), 363–396. https://doi.org/10.1145/1089014.1089020
[42] Alan C. Hindmarsh and Syvert P. Nørsett. 1988. *KRYSI, an ODE solver combining a semi-implicit Runge-Kutta method and a preconditioned Krylov method.* Technical Report. Lawrence Livermore National Laboratory, Livermore, CA (USA); Trondheim Univ. (Norway).
[43] Alan C. Hindmarsh and Linda R. Petzold. 1995. Algorithms and software for ordinary differential equations and differential-algebraic equations, Part I: Euler methods and error estimation. *Comput. Phys.* 9, 1 (1995), 34–41. https://doi.org/10.1063/1.168536
[44] Alan C. Hindmarsh and Linda R. Petzold. 1995. Algorithms and software for ordinary differential equations and differential-algebraic equations, Part II: Higher-order methods and software packages. *Comput. Phys.* 9, 2 (1995), 148–155. https://doi.org/10.1063/1.168540
[45] Alan C. Hindmarsh, Radu Serban, Cody J. Balos, David J. Gardner, Daniel R. Reynolds, and Carol S. Woodward. 2021. User Documentation for CVODE. v6.2.0.
[46] Intel. 2022. *Developer Reference for Intel oneAPI Math Kernel Library.* https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html
[47] Kenneth R. Jackson and Ron Sacks-Davis. 1980. An alternative implementation of variable step-size multistep formulas for stiff ODEs. *ACM Trans. Math. Software* 6, 3 (1980), 295–318. https://doi.org/10.1145/355900.355903
[48] Seth Johnson, Andrey Prokopenko, and Katherine Evans. 2019. Automated Fortran-C++ Bindings for Large-Scale Scientific Applications. *Comput. Sci. Eng.* PP (2019), 1–1. https://doi.org/10.1109/MCSE.2019.2924204
[49] Ian Karlin et al. 2019. Preparation and Optimization of a Diverse Workload for a Large-Scale Heterogeneous System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 32, 17 pages. https://doi.org/10.1145/3295500.3356192
[50] Christopher A. Kennedy and Mark H. Carpenter. 2003. Additive Runge–Kutta schemes for convection–diffusion–reaction equations. *Appl. Numer. Math.* 44, 1-2 (Jan 2003), 139–181. https://doi.org/10.1016/S0168-9274(02)00138-1
[51] Christopher A. Kennedy and Mark H. Carpenter. 2019. Higher-Order Additive Runge–Kutta Schemes for Ordinary Differential Equations. *Appl. Numer. Math.* 136 (Feb 2019), 183–205. https://doi.org/10.1016/j.apnum.2018.10.007
[52] Stefan J. Kollet, Reed M. Maxwell, Carol S. Woodward, Steve Smith, Jan Vanderborght, Harry Vereecken, and Clemens Simmer. 2010. Proof of concept of regional scale hydrologic simulations at hydrologic resolution utilizing massively parallel computer resources. *Water Resour. Res.* 46, 4 (2010), 7 pages. https://doi.org/10.1029/2009WR008730
[53] Shengtai Li and Linda Petzold. 2000. Software and algorithms for sensitivity analysis of large-scale differential algebraic systems. *J. Comput. Appl. Math.* 125, 1-2 (2000), 131–145. https://doi.org/10.1016/S0377-0427(00)00464-7
[54] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Software* 29, 2 (June 2003), 110–140. https://doi.org/10.1145/779359.779361

[55] Benny Malengier, Pavol Kišon, James Tocknell, Claas Abert, Florian Bruckner, and Marc-Antonio Bisotti. 2018. ODES: a high level interface to ODE and DAE solvers. *J. Open Source Softw.* 3, 22 (2018), 165. https://doi.org/10.21105/joss.00165
[56] Timothy Maly and Linda R. Petzold. 1996. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Appl. Numer. Math.* 20, 1-2 (1996), 57–79. https://doi.org/10.1016/0168-9274(95)00117-4
[57] Matthew J. McNenly, Russell A. Whitesides, and Daniel L. Flowers. 2015. Faster solvers for large kinetic mechanisms using adaptive preconditioners. *Proc. Combust. Inst.* 35, 1 (2015), 581–587. https://doi.org/10.1016/j.proci.2014.05.113
[58] NVIDIA. 2019. cuSOLVER library, release 10.1.243. https://docs.nvidia.com/cuda/archive/10.1/pdf/CUSOLVER_Library.pdf.
[59] James M. Ortega and Werner C. Rheinboldt. 2000. *Iterative solution of nonlinear equations in several variables.* SIAM Publications, Philadelphia, PA. https://doi.org/10.1137/1.9780898719468
[60] Michael Pernice and Homer F. Walker. 1998. NITSOL: A Newton iterative solver for nonlinear systems. *SIAM J. Sci. Comput.* 19, 1 (1998), 302–318. https://doi.org/10.1137/S1064827596303843
[61] Linda R. Petzold. 1982. *Description of DASSL: A differential/algebraic system solver.* Technical Report. Sandia National Laboratory, Livermore, CA (USA). https://www.osti.gov/biblio/5882821
[62] Christopher Rackauckas and Qing Nie. 2017. DifferentialEquations.jl – A performant and feature-rich ecosystem for solving differential equations in Julia. *J. Open Res. Softw.* 5, 1 (2017), 10 pages. https://doi.org/10.5334/jors.151
[63] Daniel R. Reynolds, David J. Gardner, Cody J. Balos, and Carol S. Woodward. 2019. *SUNDIALS Multiphysics+MPIManyVector Performance Testing.* Technical Report. Lawrence Livermore National Laboratory. https://doi.org/10.2172/1567994
[64] Adrian Sandu. 2019. A Class of Multirate Infinitesimal GARK Methods. *SIAM J. Numer. Anal.* 57, 5 (Jan 2019), 2300–2327. https://doi.org/10.1137/18M1205492
[65] Martin Schlegel, Oswald Knoth, Martin Arnold, and Ralf Wolke. 2009. Multirate Runge–Kutta schemes for advection equations. *J. Comput. Appl. Math.* 226, 2 (Apr 2009), 345–357. https://doi.org/10.1016/j.cam.2008.08.009
[66] Martin Schlegel, Oswald Knoth, Martin Arnold, and Ralf Wolke. 2012. Implementation of multirate time integration methods for air pollution modelling. *Geosci. Model Dev.* 5, 6 (Nov 2012), 1395–1405. https://doi.org/10.5194/gmd-5-1395-2012
[67] Martin Schlegel, Oswald Knoth, Martin Arnold, and Ralf Wolke. 2012. Numerical solution of multiscale problems in atmospheric modeling. *Appl. Numer. Math.* 62, 10 (Oct 2012), 1531–1543. https://doi.org/10.1016/j.apnum.2012.06.023
[68] Henning Schmidt. 2007. SBaddon: High performance simulation for the Systems Biology Toolbox for MATLAB. *Bioinformatics* 23, 5 (2007), 646–647. https://doi.org/10.1093/bioinformatics/btl668
[69] Radu Serban and Alan C. Hindmarsh. 2005. CVODES: The sensitivity-enabled ODE solver in SUNDIALS. In *Proceedings of the ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. (International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Vol. Volume 6: 5th International Conference on Multibody Systems, Nonlinear Dynamics, and Control, Parts A, B, and C).* American Society of Mechanical Engineers, 257–269. https://doi.org/10.1115/DETC2005-85597
[70] Lawrence F. Shampine. 1999. Conservation laws and the numerical solution of ODEs, II. *Comput. Math. with Appl.* 38, 2 (1999), 61–72. https://doi.org/10.1016/S0898-1221(99)00183-2
[71] The SUNDIALS development team. 2022. *SUNDIALS.* https://computing.llnl.gov/projects/sundials
[72] The SUNDIALS development team. 2022. *SUNDIALS GitHub.* https://github.com/LLNL/sundials
[73] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36, 5-6 (June 2010), 232–240. https://doi.org/10.1016/j.parco.2009.12.005
[74] Christopher J. Vogl, Andrew Steyer, Daniel R. Reynolds, Paul A. Ullrich, and Carol S. Woodward. 2019. Evaluation of Implicit-Explicit Additive Runge-Kutta Integrators for the HOMME-NH Dynamical Core. *J. Adv. Model. Earth Syst.* 11, 12 (2019), 4228–4244. https://doi.org/10.1029/2019MS001700
[75] Homer F. Walker and Peng Ni. 2011. Anderson acceleration for fixed-point iterations. *SIAM J. Numer. Anal.* 49, 4 (2011), 1715–1735. https://doi.org/10.1137/10078356X
[76] Wolfram. 2020. *IDA Method for NDSolve.* https://reference.wolfram.com/language/tutorial/NDSolveIDAMethod.html
[77] Weiqun Zhang, Andrew Myers, Kevin Gott, Ann Almgren, and John Bell. 2021. AMReX: Block-structured adaptive mesh refinement for multiphysics applications. *Int. J. High Perform. Comput. Appl.* 35, 6 (2021), 508–526. https://doi.org/10.1177/10943420211022811