Solution of the Modified Bratu Problem in SAMRAI

Michael Pernice* Brian T. N. Gunney†

February 26, 2004

Abstract

A model implementation of the solution of an unsteady nonlinear reaction-diffusion on a SAMR grid using SAMRAI has been developed. This model implementation illustrates the use of new capabilities for implicit timestepping and solution of large-scale systems of nonlinear equations using implementations of inexact Newton methods found in KINSOL and PETSc. This document provides a detailed description of the implementation.

1 Introduction

Adaptive mesh refinement (AMR) is a computational approach that concentrates effort in regions where it is most needed by using a finer grid only where enhanced resolution is necessary. Numerous AMR algorithms for hyperbolic problems with explicit timestepping and some classes of elliptic problems have been described. However more general nonlinear problems that may also involve implicit timestepping have not received as much attention.

Structured adaptive mesh refinement (SAMR) is an AMR strategy in which the computational mesh is organized as a hierarchy of nested levels, each of which is a union of rectangular regions. SAMRAI is a software framework developed at LLNL/CASC that supports the use of SAMR methods in a wide variety of problems in computational science and engineering. SAMRAI includes sophisticated numerical algorithms that provide complete implementations for some classes of problems, such as systems of hyperbolic conservation laws. Interfaces to other high quality numerical libraries, such as hypre, KINSOL, and PETSc, either exist or are under development. Support for implicit timestepping has also recently been completed.

Model applications are being created in concert with the development of these new capabilities. These applications serve to fill out the functionality of the new capabilities, test the implementations, and demonstrate the viability

*Computer and Computational Sciences Division, Los Alamos National Laboratory
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory

of the ideas behind the new capabilities. The model applications also serve as demonstration codes that other application developers can use to create software to solve other problems. This document describes one such model application in detail.

This document is organized as follows. Model problem specification, including discretization on a SAMR grid, is discussed in §2. This discussion is purposefully generic, in order to motivate the software requirements that follow. A mapping of this problem specification onto SAMRAI data structures appears in §3. Interfaces to implementations of user-defined methods for implicit timestepping and the nonlinear solvers are described in §4.

2 Problem Specification

The problem solved in this model application is a modified version of the nonlinear Bratu problem:

$$\frac{\partial u}{\partial t} = \nabla \cdot D\nabla u + \lambda e^u + f(\mathbf{x}, t, u), \qquad t \geq 0, \qquad \mathbf{x} = (x, y, z) \in \Omega \qquad (1)$$

where $\Omega = [0, 1]^3$. Here, the diffusion coefficient D can depend on space and time.[1] There are three features in (1) that are not present in the classical Bratu problem:

1. the classical Bratu problem is a steady-state problem;
2. the classical Bratu problem has uniform diffusivity;
3. the source term f is not present in the classical Bratu problem.

This latter feature was added to allow the solution u to be specified so that correctness of the implementation and accuracy of the solution could be assessed.

Specification of the problem is completed with initial and boundary conditions. Although general initial and boundary values are possible, only $u(\mathbf{x}, 0) = 0$ and $u(\mathbf{x}, t) = 0, \mathbf{x} \in \partial\Omega$ are implemented.

A SAMR grid may be represented as a grid hierarchy. The hierarchy consists of a number of levels, with level 0 indicating the coarsest level. Each level consists of a union of rectangular regions, or patches, at the same resolution. This hierarchical representation allows operations on the hierarchy to be decomposed into operations on individual patches.

Cells in all levels are referenced relative to a global index space. Thus, level 0 is referenced relative to the base discretization: if level 0 is an $n_x \times n_y \times n_z$ grid, then cells are indexed according to $i = 0, \ldots, n_x - 1$, $j = 0, \ldots, n_y - 1$, and $k = 0, \ldots, n_z - 1$. If a refinement ratio of 2 is used, then level 1 is indexed according to $i = 0, \ldots, 2n_x - 1$, $j = 0, \ldots, 2n_y - 1$, and $k = 0, \ldots, 2n_z - 1$. Depending on how the level 0 grid is refined, only a subset of these indices actually represents the solution.

[1] It can also depend on u, but this is not implemented.

An example of a grid hierarchy appears in Figure 1, which shows a SAMR grid with three levels and two patches on each of the two refinement levels. The coarsest level is defined on an index space with lower left corner $(0,0)$ and upper right corner $(7,7)$. The first refinement level is defined on index space with lower left corner $(0,0)$ and upper right corner $(15,15)$, and consists of two rectangular patches. One has lower left corner $(2,4)$ and upper right corner $(7,9)$ and the other has lower left corner $(8,6)$ and upper right corner $(13,15)$. Finally the finest level is defined on an index space with lower left corner $(0,0)$ and upper right corner $(31,31)$ and also consists of two patches. One has lower left corner $(8,14)$ and upper right corner $(21,17)$ and the other has lower left corner $(18,18)$ and upper right corner $(25,31)$.

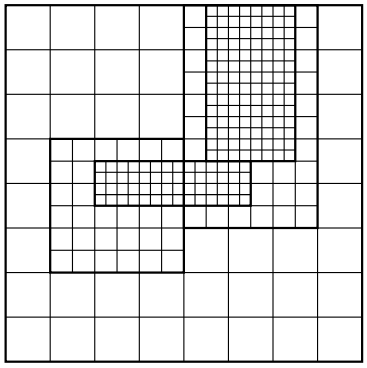

Figure 1: Example of a multilevel SAMR grid with three levels.

The solution on a grid hierarchy is defined at spatial locations that correspond to the finest available grid. Thus, all cells on the finest grid are part of the solution. All cells on the next coarser level that have not been refined to the next finer level are also part of the solution. This process continues until level 0 is reached. At the risk of being less precise, another way to say this is that the solution is defined only in cells that have not been refined. Values defined in cells that have been refined only serve to represent the level as a union of rectangles, and are usually defined as averages of values on the next finer grid. These cells can also be used to accelerate the convergence of iterative solution procedures in a manner similar in spirit to multigrid methods.

Discretization on a grid with a single resolution is described first. Following this, modifications for extending the discretization to a grid hierarchy are discussed. The section concludes with a description of how boundary conditions may be treated.

2.1 Single grid discretization

Implicit discretization of (1) in time and rearrangement leads to

$$u^{(n+1)} - u^{(n)} - \Delta t \left(\nabla \cdot D \nabla u^{(n+1)} + \lambda e^{u^{(n+1)}} + f(\mathbf{x}, t_{n+1}, u^{(n+1)}) \right) = 0$$

where Δt is the current time step and superscript (n) refers to the n-th timestep. Thus the solution is advanced in time by solving the nonlinear equation

$$F(u; u^{(n)}) = u - u^{(n)} - \Delta t \left(\nabla \cdot D \nabla u + \lambda e^{u} + f(\mathbf{x}, t_{n+1}, u) \right) = 0. \tag{2}$$

A discrete problem is obtained by subdividing Ω into cells Ω_{ijk} and seeking the solution in the discrete form

$$u_{ijk}^{(n)} \approx u(x_i, y_j, z_k, t_n)$$

where (x_i, y_j, z_k) is the center of Ω_{ijk}. Integrating (2) over Ω_{ijk} gives

$$\begin{aligned}
F_{ijk} &\equiv \int_{\Omega_{ijk}} F(u; u^{(n)}) \, dV \\
&= \int_{\Omega_{ijk}} (u - u^{(n)}) \, dV && (3) \\
&\quad -\Delta t \int_{\Omega_{ijk}} \nabla \cdot D \nabla u \, dV && (4) \\
&\quad -\Delta t \int_{\Omega_{ijk}} \lambda e^{u} \, dV && (5) \\
&\quad -\Delta t \int_{\Omega_{ijk}} f(\mathbf{x}, t_n, u) \, dV. && (6)
\end{aligned}$$

The integrals (3), (5) and (6) are approximated by

$$\begin{aligned}
\int_{\Omega_{ijk}} (u - u^{(n)}) \, dV &\approx (u_{ijk} - u_{ijk}^{(n)}) \Delta V, && (7) \\
\int_{\Omega_{ijk}} \lambda e^{u} \, dV &\approx \lambda e^{u_{ijk}} \Delta V, && (8) \\
\int_{\Omega_{ijk}} f(\mathbf{x}, t, u) \, dV &\approx f(x_i, y_j, z_k, t_n, u_{ijk}) \Delta V, && (9)
\end{aligned}$$

where $\Delta V \equiv \Delta x \Delta y \Delta z$ is the volume of Ω_{ijk}.

The integral (4) is approximated by appealing to the divergence theorem:

$$\begin{aligned}
\int_{\Omega_{ijk}} \nabla \cdot D \nabla u \, dV &= \int_{\partial \Omega_{ijk}} D \nabla u \cdot \mathsf{n} \, dA \\
&\approx \left(D_{i+\frac{1}{2},j,k} (u_{i+1,j,k} - u_{i,j,k}) - D_{i-\frac{1}{2},j,k} (u_{i,j,k} - u_{i-1,j,k}) \right) \frac{\Delta y \Delta z}{\Delta x} \\
&\quad + \left(D_{i,j+\frac{1}{2},k} (u_{i,j+1,k} - u_{i,j,k}) - D_{i,j-\frac{1}{2},k} (u_{i,j,k} - u_{i,j-1,k}) \right) \frac{\Delta x \Delta z}{\Delta y} \\
&\quad + \left(D_{i,j,k+\frac{1}{2}} (u_{i,j,k+1} - u_{i,j,k}) - D_{i,j,k-\frac{1}{2}} (u_{i,j,k} - u_{i,j,k-1}) \right) \frac{\Delta x \Delta y}{\Delta z} \qquad (10)
\end{aligned}$$

where n is the unit outward-facing normal to $\partial\Omega_{ijk}$.

The full discrete system is the sum of (7), (10), (8) and (9). This is not written down here explicitly in the interest of saving space.

Remark 2.1 Observe that the discrete system may be written

$$F(u; u^{(n)}) = Au + \lambda e^u + f(\mathbf{x}, t, u)$$

where A is a matrix with seven nonzero diagonals (for the 3D case). Using stencil notation, the east, west, north, south, top, bottom, and center weights are given respectively by

$$\begin{aligned}
ae_{i,j,k} &= D_{i+\frac{1}{2},j,k}\frac{\Delta y \Delta z}{\Delta x}, \\
aw_{i,j,k} &= D_{i-\frac{1}{2},j,k}\frac{\Delta y \Delta z}{\Delta x}, \\
an_{i,j,k} &= D_{i,j+\frac{1}{2},k}\frac{\Delta x \Delta z}{\Delta y}, \\
as_{i,j,k} &= D_{i,j-\frac{1}{2},k}\frac{\Delta x \Delta z}{\Delta y}, \\
at_{i,j,k} &= D_{i,j,k+\frac{1}{2}}\frac{\Delta x \Delta y}{\Delta z}, \\
ab_{i,j,k} &= D_{i,j,k-\frac{1}{2}}\frac{\Delta x \Delta y}{\Delta z}, \\
ac_{i,j,k} &= -(at_{i,j,k} + ab_{i,j,k} + an_{i,j,k} + as_{i,j,k} + ae_{i,j,k} + aw_{i,j,k}).
\end{aligned}$$

Remark 2.2 Observe that the Jacobian of the discrete system is

$$F'(u; u^{(n)}) = A + \lambda e^u + \frac{\partial f}{\partial u}(\mathbf{x}, t, u). \tag{11}$$

Remark 2.3 For discretization on a SAMR grid, it is useful to regard the six contributions in the approximation of $\int_{\Omega_{ijk}} \nabla \cdot D\nabla u \, dV$ as fluxes of u across the respective faces of Ω_{ijk}.

2.2 Multilevel discretization

Except for cells at interfaces between coarse and fine levels, the problem can be discretized on each level exactly as described in §2.1. Of the four contributions to the discrete version of the problem, (3), (5) and (6) can be approximated with (7), (8) and (9) respectively. Only the approximation of (4) needs modification, and this modification only needs to be made on the sides of Ω_{ijk} that are adjacent to one or more cells that reside at a different level of resolution. This situation, using a two-dimensional example, is depicted in Figure 2.

Consider first the fine side of the interface. Data from the coarse level is needed to define the flux on the west side of fine cells. This can be accomplished by interpolating data at the coarser resolution to a location in a ghost cell at

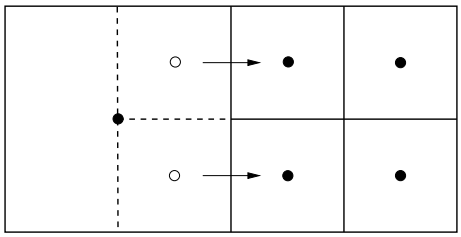

Figure 2: Compuational cells at an interface between coarse and fine levels.

the finer resolution. This interpolation can be done in a variety of ways, with the caveat that the choice affects the accuracy of the solution in the fine region. For example, linear interpolation on the coarse side of the interface leads to the discretization of [1, 2]. Once the fluxes are defined, the approximation (10) can be computed by differencing the fluxes over the cells in the usual way.

Consider next the coarse side of the interface. Since coarse data is defined in the coarse cell underlying the four fine cells, calculation of fluxes on the coarse level can proceed in the usual fashion. However, the flux computed at the east face of the coarse cell will not in general be equal to the sum of the fluxes computed at the west faces of the adjacent fine cells. To correct this, the flux at the east side of the coarse cell is replaced by the sum of the fluxes on the west sides of the fine cells. More precisely,

$$\int_{\partial\Omega^{2h}_{i_c+\frac{1}{2},j_c}} Du_x \, dA = \int_{\partial\Omega^{h}_{i_f-\frac{1}{2},j_f}} Du_x \, dA + \int_{\partial\Omega^{h}_{i_f-\frac{1}{2},j_f+1}} Du_x \, dA,$$

where (i_c, j_c) is the index of the coarse cell and (i_f, j_f), $(i_f, j_f + 1)$ are indices of the two fine cells adjacent to the coarse/fine interface. The notations h and $2h$ respectively denote the fine and coarse meshes. Extension to three dimensions, in which four fine face fluxes must be accumulated, is straightforward.

Once the flux is computed and corrected on the faces of the control volumes of the coarse levels, the approximation (10) can be computed by differencing the fluxes over the cells in the usual way.

While this description lacks complete detail, the general nature of what needs to be done at coarse/fine interfaces is clear. Summarizing:

- on the fine side of the coarse/fine interface, ghost cell information must be interpolated from the next coarser level;
- on the coarse side of the coarse/fine interface, fluxes on interfaces adjacent to the fine level must be matched with the sum of the fluxes from adjacent fine cell faces.

Remark 2.4 SAMRAI provides a set of predefined `CoarsenOperators` and `RefineOperators` that can be used to perform the needed interpolations. User-defined averaging operations can also be employed.

Remark 2.5 Using data from the coarse grid as Dirichlet boundary conditions on the next finer level ensures continuity of the solution across the coarse/fine interface. The flux matching process prevents spurious sources at coarse/fine interfaces.

2.3 Treatment of boundary conditions

The discussion of the multilevel discretization revealed the need for ghost cells, where they were used to provide storage for data interpolated from the next coarser level resolution. They also can be used to provide storage for data at the same level of resolution that resides on a different patch. Finally, they can be used to provide a uniform mechanism for treating physical boundary conditions, which can be implemented in a variety of ways. With boundary values defined on cell faces that reside at the physical boundary, an extrapolation procedure can be used to compute values that are placed in ghost cells; this has the disadvantage that every time the solution is changed in a cell adjacent to the physical boundary, the ghost cell value must also be changed. Alternatively, the computation of the flux on the physical boundary can be adjusted to reflect the different grid spacing; this has the disadvantage of dropping the order of the discretization by one at the physical boundary. Another alternative is to compute the flux at the physical boundary by fitting a quadratic normal to the physical boundary and taking its derivative at the physical boundary; this can handle either Neuman or Dirichlet boundary conditions and preserves the order of the discretization at the physical boundary, but requires data from two cells in the direction normal to the boundary.

3 Mapping onto SAMRAI data structures

Special care is needed to design software for solving problems on SAMR grids. While it is straightforward to map data defined on a single, global grid onto linear memory space, there is no clear way to do this for a multilevel grid hierarchy. Further, SAMR grids can change dynamically during the course of the computation. Implementation for distributed memory computer architectures adds another degree of complexity.

One way to address this situation is to separate the concept of a variable from the associated storage. SAMRAI allows users to specify variables in a manner that is independent of the associated storage. Data on individual patches is mapped to linear memory space lexicographically, and lookup mechanisms provide access to this storage on a per-level, per-patch basis. The decomposition of grid levels into patches maps naturally onto distributed memory, by assigning each patch to a single processor. In addition, SAMRAI provides a means to associate multiple usage contexts with each variable. This simplifies naming conventions and allows storage associated with variables to be managed in groups. Finally, a uniform mechanism for moving data among patches at the same or different levels of resolution is provided.

3.1 Variables

The variables and grid centerings needed are listed in Table 1. The solution, nonlinear term, and artificial source are all cell-centered variables. The fluxes that must be computed reside on cell faces. They can be represented with either a `FaceVariable` or a `SideVariable`; since the former uses a permuted index convention, the latter was chosen. Note that the `SideVariable`s encapsulate data on all sides of the cells, and have d components (where d is the dimension of the problem), corresponding to faces normal to each of the coordinate directions.

Quantity	Role	Type	Name
u	solution	`CellVariable`	`d_solution`
D	diffusion coefficient	`SideVariable`	`d_diffusion_coef`
$D\nabla u$	fluxes	`SideVariable`	`d_flux`
λe^u	nonlinear term	`CellVariable`	`d_exponential_term`
f	artificial source	`CellVariable`	`d_source_term`

Table 1: List of variables.

One additional variable that does not appear on this list is needed. As noted in §2.2, once fluxes on a coarse level are computed, the values that reside at coarse/fine interfaces must be replaced by the sum of the fine fluxes from adjacent cells at the next finer level. Fluxes from the finer level can be computed and saved in an `OutersideVariable`, which resides only on the outer faces of patches. This data can subsequently be averaged to the corresponding locations on the next coarser level. These variables have $2d$ components, corresponding to two sides in each of the coordinate directions. This variable is labeled `d_coarse_fine_flux`.

3.2 Contexts

During the course of a simulation, some variables must play multiple roles. For example, in an unsteady problem, storage for the solution at the previous, current, and next timesteps might be used. Some storage is used only for scratch space, playing a role only during intermediate stages of a calculation. Common practice is to use more descriptive names: for example multiple timesteps might be labeled `u_old`, `u`, and `u_new`. SAMRAI provides `VariableContext`s for this purpose. Table 2 lists the contexts used in this model application.

Actual storage is not associated with variables alone; data is allocated using a variable-context pair. This pair is uniquely associated with a single integer descriptor index that is maintained in a `VariableDatabase`. Actual storage is associated with each patch in the SAMR grid together with a variable-context pair (or, equivalently, with a patch together with a descriptor index).

As noted in §2.3, ghost cells provide a convenient and uniform mechanism for treating off-patch data dependencies. The need for ghost cells affects the amount of storage needed for a (variable, context). Ghost cells must be used judiciously to avoid unnecessary storage overhead. Here, only the scratch context of the

Variable	Context	Ghost Cells	Role
d_solution	current	0	holds the current time step
d_solution	new	0	holds the next time step
d_solution	scratch	1	scratch space for computation
d_diffusion_coef	scratch	0	intermediate quantity
d_flux	scratch	0	intermediate quantity
d_exponential_term	scratch	0	intermediate quantity
d_source_term	scratch	0	intermediate quantity
d_coarse_fine_flux	scratch	0	intermediate quantity
d_jacobian_a	scratch	0	intermediate quantity
d_jacobian_b	scratch	0	intermediate quantity
d_precond_a	scratch	0	intermediate quantity
d_precond_b	scratch	0	intermediate quantity

Table 2: List of contexts.

d_solution variable has a layer of ghost cells. The roles played by this particular variable-context pair are described in §§3.3 and 4.3.1.

3.3 Data movement

Data must be moved among different storage locations at various stages of the calculation. For example, ghost cells in the approximate solution must be filled with values from neighboring patches at the same level of resolution. Fluxes that reside at coarse cell faces at coarse/fine interfaces must be replaced by the sum of the fluxes from adjacent fine cell faces.

Data movement is managed in SAMRAI by maintaining lists of the quantities that must be moved and the pattern of data movement that is dictated by the geometry of the grid hierarchy in separate data structures. This allows a single specification of the quantities that must be transferred, as well as flexibility in changing the pattern of data transfers as the hierarchy changes. The quantities that must be moved are registered with CoarsenAlgorithms and RefineAlgorithms (the former moves data from a level to a coarser level; the latter moves data from a level to one at the same or finer resolution). The pattern of movement is represented by CoarsenSchedules and RefineSchedules. Suitable CoarsenOperators and RefineOperators are registered with these schedules. Data transfer members that are used throughout the model application are listed in Table 3.

Several other transfer algorithms are used locally during the calculation. These are described in §4.3.1.

4 Interfaces to implicit timestepping and nonlinear solvers

New capabilities for performing implicit timestepping and solving systems of nonlinear equations have been added to SAMRAI. These packages are designed

Transfer Algorithm	Source	Destination	Role
`d_fill_new_level`	current solution	current solution	initialize data after regridding
`d_flux_coarsen`	flux	coarse/fine flux	correct fluxes on faces of coarse cells the coarse/fine interfaces
`d_soln_fill`	nonlinear solver soln	scratch soln	Update ghosts before before computation
`d_soln_coarsen`	nonlinear solver soln	nonlinear solver soln	Overide coarse soln with fine soln
`d_scratch_soln_coarsen`	scratch soln	scratch soln	Overide coarse soln with fine soln in scratch space

Table 3: Action of various transfer algorithms. Corresponding schedules are created from these.

to leverage existing high-quality implementations of inexact Newton methods found in KINSOL and PETSc. At the same time, care is being taken to make the interfaces general and flexible enough to accommodate custom user approaches. In the future, additional interfaces to nonlinear multigrid methods will be developed.

Our example uses the backward Euler implicit time discretization. This serves as a useful archetype for exploring the issues needed to create general and flexible capabilities. Support for higher order time discretization will follow. Our outline for the backward Euler method appears in Table 4

Procedure BEM: Backward Euler Method

SET $t = t_0$, $n = 0$, $\mathbf{u}^{(0)} = \mathbf{u}_0$.
CHOOSE AN INITIAL Δt.
DO {
  DO {
    CHOOSE AN INITIAL APPROXIMATION FOR $\mathbf{u}^{(n+1)}$.
    SOLVE $F(\mathbf{u}; \mathbf{u}^{(n)}) = 0$.
    IF ($u^{(n+1)}$ IS NOT SATISFACTORY) {
      REDUCE Δt
    }
  } (UNTIL $u^{(n+1)}$ IS SATISFACTORY.)
  UPDATE SOLUTION.
  $t = t + \Delta t$, $n = n + 1$.
  CHOOSE A NEW Δt.
} (WHILE $t \leq t_{final}$.)

Table 4: Outline of backward Euler method.

Figure 3 depicts the general organization of relations among the various com-

ponents. The `ImplicitIntegrator` orchestrates the timestepping procedure. It calls a number of methods to advance the solution in time. These are organized into methods for treating the spatial and temporal discretization.

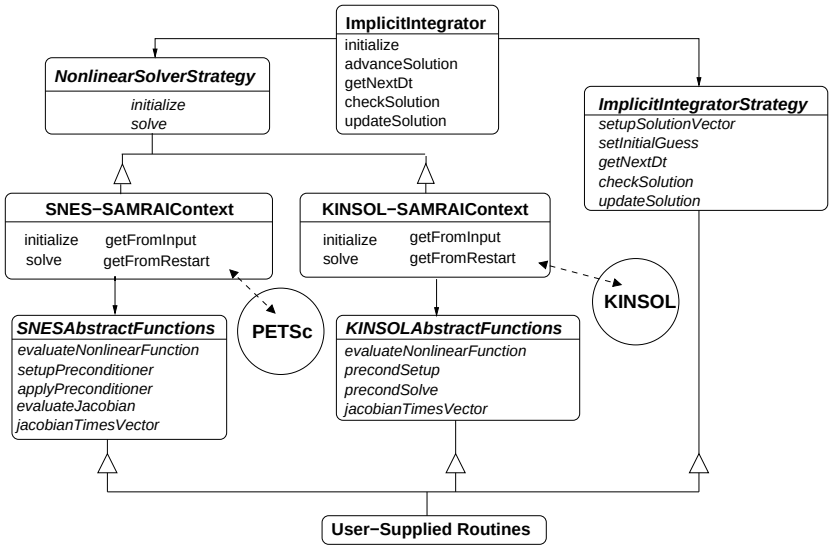

Figure 3: Organizational structure of implicit timestepping and nonlinear solver interfaces.

Methods for treating temporal discretization are defined in the `ImplicitIntegratorStrategy`. This is a pure abstract class that only defines interfaces; the implementations must be provided by the user.

Interfaces for solving the system of nonlinear equations are defined in the `NonlinearSolver`. This is also a pure abstract class that only defines interfaces. Interfaces to two concrete nonlinear solvers have been written: KINSOL and the SNES package in PETSc. These both implement inexact Newton methods. Interfaces to required and optional user-supplied methods are specified in the `AbstractFunctions` classes.

Leveraging KINSOL and PETSc was facilitated by the fact that both are written in terms of operations on vectors. This was accomplished by providing a definition of vectors on SAMR grids, together with a suitable set of operations that act directly on storage locations managed by SAMRAI. These vectors also provide a natural way to apply the implicit integrator to problems that involve more than one variable each gridpoint.

The following sections provide more detailed discussion of the interfaces to user-provided methods. First, a general notion of vectors on SAMR grids, and their specialization to KINSOL and PETSc vectors, is discussed. Next, the interfaces defined in the `ImplicitIntegratorStrategy` are discussed. Finally, the interfaces defined in the `NonlinearSolver` and `AbstractFunctions` are

described.

4.1 SAMRAI vectors

Typically, vectors are used as containers for any number of variables that possibly have different grid centerings. In the simplest use, a single variable defined on a single global grid is mapped to a single-indexed vector using some ordering scheme; lexicographical ordering is the usual choice. When multiple variables are present in a simulation, a variety of mappings are possible. For example, storage can be mapped to single-index vector locations one variable at a time, or one grid cell at a time (sometimes referred to as block mappings). When mixed centerings are used, as happens, for example, in a staggered grid discretization for fluids in which the velocity field is face-centered and the pressure is cell-centered, a bit more care is needed due to the different number of grid locations for each type of grid centering.

The concept of a vector gets stretched a bit in a distributed memory environment. Additional bookkeeping for mapping portions of vectors onto processors is needed. Often the notion of a global index as well as local indices is employed. Nevertheless operations can readily be defined on such distributed vectors without difficulty.

Note that it is often the case that a dual description of the data is maintained in a code, if only implicitly. In one description the data resides in contiguous single-indexed locations. In another the grid-based nature of the data is used through references to nearest neighbors using offsets into other single-indexed locations. In fully unstructured calculations this idea is generalized by supplementing the vector representation with neighbor lists associated with each vector index.

A SAMR grid introduces new complications. For one thing, not all grid locations on all grid levels are part of the solution. For another, the grid can change dynamically during the course of the calculation. This is handled in SAMRAI through a `SAMRAIVector`, which is simply a container analogous to the usage described in the preceding paragraphs. Different variable-context pairs are registered as components of the vector. A pointer to the grid hierarchy over which the variables are defined is included in a `SAMRAIVector`. Vector operations that are suitable for the different possible types of grid centerings, along with appropriate lookup operations, are also provided. Recall from the discussion in §3.2 that actual storage locations require both the variable-context pair together with a specific patch from a level in the grid hierarchy; all of this information is present within a `SAMRAIVector`. This structure is depicted in Figure 4.

It is also possible to register a weight vector that applies to one or more components that are registered with the `SAMRAIVector`. This weight can be used to serve several purposes. For one, it can be used to mask data on coarse levels that are covered by a finer level. It can also be used to store quadrature information that reflects the manner in which the discretization is derived. For example in the discretization derived in §2, it is natural to define a norm by

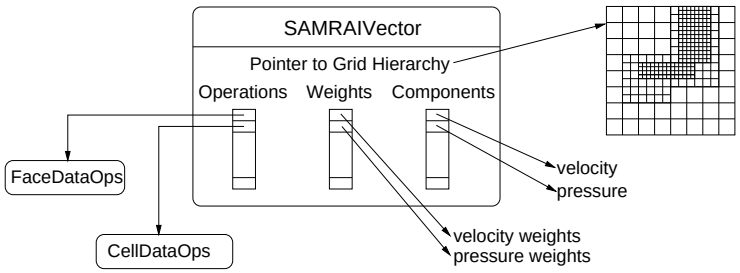

Figure 4: Layout of a `SAMRAIVector`.

summing over the levels in the hierarchy, patches in each level, and indexes in each patch:

$$\|f\|^2 = \frac{1}{V} \sum_{\mathcal{L} \in \mathcal{H}} \sum_{\mathcal{P} \in \mathcal{L}} \sum_{(i,j,k) \in \mathcal{P}} |f_{i,j,k}|^2 V_{i,j,k}$$

where V is the volume of the domain Ω and

$$V_{i,j,k} = \begin{cases} \text{volume of } \Omega_{i,j,k} & \text{if cell } (i,j,k) \text{ is not covered by a finer cell} \\ 0 & \text{otherwise} \end{cases}$$

This definition of a norm has the property that the norm of a constant is independent of the grid. This can be accomplished by a suitable initialization of a weight vector.

KINSOL `N_vectors` and PETSc `Vecs` are distinct and have different operations defined that are determined by the way they are used in their respective solver packages. A SAMRAI version of these are obtained by wrapping a `SAMRAIVector` to provide the needed interfaces. This is illustrated in Figure 5. There is minimal additional storage overhead, and maintenance is simplified.

There is one final point worth noting. Not all variables require ghost cells. `KINSOL` and `PETSc` both create internal workspace by cloning vectors specified by the user. Including ghost cells in vectors could lead to an unacceptable storage overhead. Also, embedding ghost cells in a vector could reduce the efficiency of vector operations by requiring non-unit strides between vector elements separated by ghost cells. This choice has consequences for how user-defined operations must be implemented and are discussed in §4.3

4.2 Interfaces for controlling timestepping

Interfaces for controlling timestepping are defined in `ImplicitIntegratorStrategy`. The intention is to provide the user with complete control over how the solution is advanced in time. Consequently this is a pure abstract class with no implementations. In the future, some implementations that can be overridden will be considered.

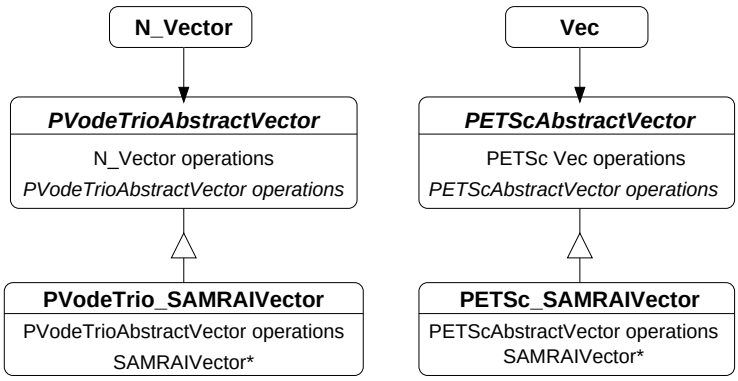

Figure 5: A `SAMRAIVector` is wrapped to provide interfaces and operations specific to `KINSOL` and `PETSc`. `KINSOL` uses the `N_Vector` objects which are `typedef`'d to `PVodeTrioAbstractVector*`. The `PVodeTrioAbstractVector` class provides the translation between `N_Vector` operations and C++ interfaces. The wrapper class `PVodeTrio_SAMRAIVector` matches that interface. The PETSc solver interfaces works similarly.

4.2.1 `setupSolutionVector`

Here the user registers all variables that comprise the solution of the problem. In this example only one component, `d_solution`, is registered.

4.2.2 `setInitialGuess`

If Δt is small enough, the solution at the current time may provide a good initial approximation for the next nonlinear solve. Alternatively, given some information from previous timesteps, it may be useful to extrapolate solution information in time to compute a new initial approximation. To accomplish this, one or more new contexts would need to be introduced and managed properly in `updateSolution`.

In this example, the solution at the current time is supplied as an initial approximation for the time-advanced solution. Implementation of a more sophisticated approach is deferred.

4.2.3 `checkSolution`

Once the nonlinear solver returns, it must be determined whether or not the solution should be accepted. For example, if the nonlinear solver failed to converge, the new solution should be rejected. Possible actions include reducing Δt and trying again, or shutting down the computation. If the nonlinear solver converges, the new solution should be tested to determine whether the temporal error is acceptable. If not, the solution should be rejected, Δt should be reduced, and a new timestep should be attempted.

In this example, all solutions returned by the nonlinear solver are accepted. Implementation of a more sophisticated approach is deferred.

4.2.4 `updateSolution`

Once a solution is accepted, storage locations for the solution at various time levels needs to be manipulated. For example, the current solution should be discarded, and the new solution copied into storage vfor the current solution. If more than one previous timestep is maintained, then the list of timesteps must be managed accordingly.

In this example, only the current and new solution are maintained.

4.2.5 `getNextDt`

Δt for the next timestep should be computed here. In this example, the initial timestep that was set through an input file is returned. A more sophisticated strategy might estimate the magnitude of the second derivative in time and use this together with a local error tolerance to set Δt.

4.3 Interfaces to the nonlinear solvers

Both KINSOL and SNES are organized so that they can be used by providing methods with prescribed signatures that implement the following operations:

- evaluation of the nonlinear function;
- evaluation of the Jacobian;
- evaluation of Jacobian-vector products;
- initialization of a preconditioner;
- application of a preconditioner.

Of these, only the first is absolutely required.

These packages also both provide a wide range of options that can be used to tune the performance of the nonlinear solver. These options can be set through an input file or through a collection of access functions.

Note that KINSOL and SNES have slightly different approaches to prescribing these interfaces. KINSOL does not prescribe an interface for evaluating the Jacobian; rather, an argument in the signature of the function for calculating Jacobian-vector products indicates whether the Jacobian should be re-evaluated.

In the interest of brevity, only the methods needed by KINSOL will be described. The methods needed by SNES are quite similar. Also, the full signatures of the operations are not used in the subsection titles; they can be found by inspecting the header files of the implementation. Briefly, the interfaces prescribed by KINSOL have been preserved. The only exception is the argument that serves as a pointer to the user's data. Since these functions are implemented in the user class, all user data is available because of the scoping rules in C++.

4.3.1 `evaluateNonlinearFunction(x,r)`

The task here is to compute the discrete form of $F(u; u^{(n)})$ in (2) on the multilevel grid. A general outline of how this is done appears in Table 5. In the implementation, details of the data and information lookup might tend to obscure the simplicity of this outline.

```
for each level in the hierarchy, finest to coarsest {
  for each patch in the level {
    compute face fluxes in (10)
    adjust computation at physical boundaries
  }
  if not on the coarsest level {
    save fluxes in an OutersideVariable
  }
  if not on the finest level {
    copy saved fluxes from finer level to cell faces at coarse/fine interfaces
  }
  for each patch in the level {
    evaluate the exponential term
    evaluate the source term
    combine all contributions (7), (8), and (9), including differencing
      the fluxes to complete evaluation of (10)
  }
}
```

Table 5: Outline of nonlinear function evaluation.

Before operating on any data, it is important to note that the input argument `x` has no ghost cells, and must be supplied with values to satisfy off-patch data dependencies. This must be done with transfer operations similar to those discussed in §3.3. However, the argument `x` could in fact be any vector created by the nonlinear solver. Thus, it is impossible to decide outside of this function what storage should serve as the source of the transfer. Thus, a `RefineAlgorithm` is created that designates `x` as the source of the transfer and the scratch solution as the destination (recall from Table 2 in §3.2 that the scratch storage context for the solution variable has one layer of ghost cells). Then, in a loop over levels in the hierarchy, this `RefineAlgorithm` is used to reset a cached `RefineSchedule` on each level, and the indicated data transfer is performed using `fillData`. (The cached `RefineSchedule` describes the data movement for a similar `RefineAlgorithm` on the current hierarchy configuration and is recreated each time the hierarchy changes.) This causes both the interior of the solution scratch storage to be copied from `x`, as well as ghost cells of the solution scratch storage to be filled on each patch.

It is worth making a few further observations on the outline that appears in Table 5.

1. All evaluation of fluxes and integrals is done in Fortran computational routines. Pointers to storage locations are retrieved using appropriate

access functions. These computations can actually be done in any language, provided that the geometry of the patch, presence of ghost cells, and gridpoint ordering conventions are all properly accounted for.

2. While not included in this outline, it may be necessary to adjust fluxes computed on the fine side of coarse/fine faces. This depends on the choices used to refine data to ghost cells on the fine level and how fluxes on the fine side of the coarse/fine faces are computed. Obtaining an index space description of the *level edge* (the set of indices of fine cells that reside at coarse/fine interfaces) is relatively straightforward to accomplish using a combination of index shifts and boolean set complement operations.

3. Recall the discussion of §2.3. Here the adjustment of the fluxes to account for boundary values is done in a separate step. While it is possible to perform this adjustment in a fixup loop inside the function that evaluates face fluxes, geometric information that describes which part of the patch is adjacent to a physical boundary is not readily available. After the flux is evaluated, the needed geometric information can be retrieved using functionality provided by SAMRAI and the adjustment of fluxes along physical boundaries can be done in a separate function call. It is also possible to perform these adjustments directly in C++ by using iterators over suitable patch indices.

4. To conserve memory, the storage associated with the fluxes is allocated and dealloacted level by level.

5. Some care is needed to similarly manage the storage for the `OutersideVariable` on the fly: it is allocated on all but the finest level just prior to being copied to, and deallocated on all levels right after being copied from.

6. The actual transfer of data to correct fluxes at coarse/fine interfaces is done with the `d_flux_coarsen`, discussed in §3.3. Since the grid hierarchy can change during the course of the calculation, the corresponding `CoarsenSchedule` is recreated each time the hierarchy changes.

4.3.2 `jacobianTimesVector(v,Jv)`

This operation is almost identical to `evaluateNonlinearFunction`, so it is not described in detail. The main difference lies in the final assembly: there the exponential term has been evaluated at the current nonlinear iterate and multiplies the input vector `v`. Considerations that applied to `x` in `evaluateNonlinearFunction` also apply here to `v`, i.e., operations are performed on a copy of `v` that has been supplemented by ghost cells.

4.3.3 `precondSetup()`

In this application, the Jacobian (11) is essentially a generalized Poisson operator. The FAC Poisson solver provided with SAMRAI is designed for precisely

this kind of problem. Thus, setup amounts to setting up and initializing the data structures needed by the Poisson solver.

Specifically, problems of the form

$$-\nabla(b\nabla u) + au = f$$

can be solved, where a is a cell-centered array, b is a face-centered array, and u and f are cell-centered arrays. In this case, $b = D$ from (1) and $a = \lambda e^{u_k}$, where u_k is the current inexact Newton iterate. Setup consists of initializing the appropriate variables and registering them with the FAC Poisson solver.

4.3.4 `precondSolve(r,z)`

Here the task is to solve

$$Mz = r.$$

The right hand side is registered with the FAC Poisson solver, which is then used to solve the preconditioning system. Note that the scratch solution, which is equipped with ghost cells, is used for this solve. The output from the solve is copied to the appropriate argument upon completion of the solve. Ghost cell data is stripped from the solution in this copy.

5 Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence-Livermore National Laboratory under contract No. W-7405-Eng-48 and by University of California Los Alamos National Laboratory under contract number W-7405-Eng-36. Document UCRL-TR-202155 and LAUR-04-0992.

References

[1] R. E. Ewing, R. D. Lazarov, and P. S. Vassilevski. Local refinement techniques for elliptic problems on cell–centered grids. I: Error analysis. *Math. Comp.*, 56:437–461, 1991.

[2] R. E. Ewing, R. D. Lazarov, and P. S. Vassilevski. Local refinement techniques for elliptic problems on cell–centered grids. II: Optimal order two-grid iterative methods. *Numer. Linear Algebra with Appl.*, 1:337–368, 1994.