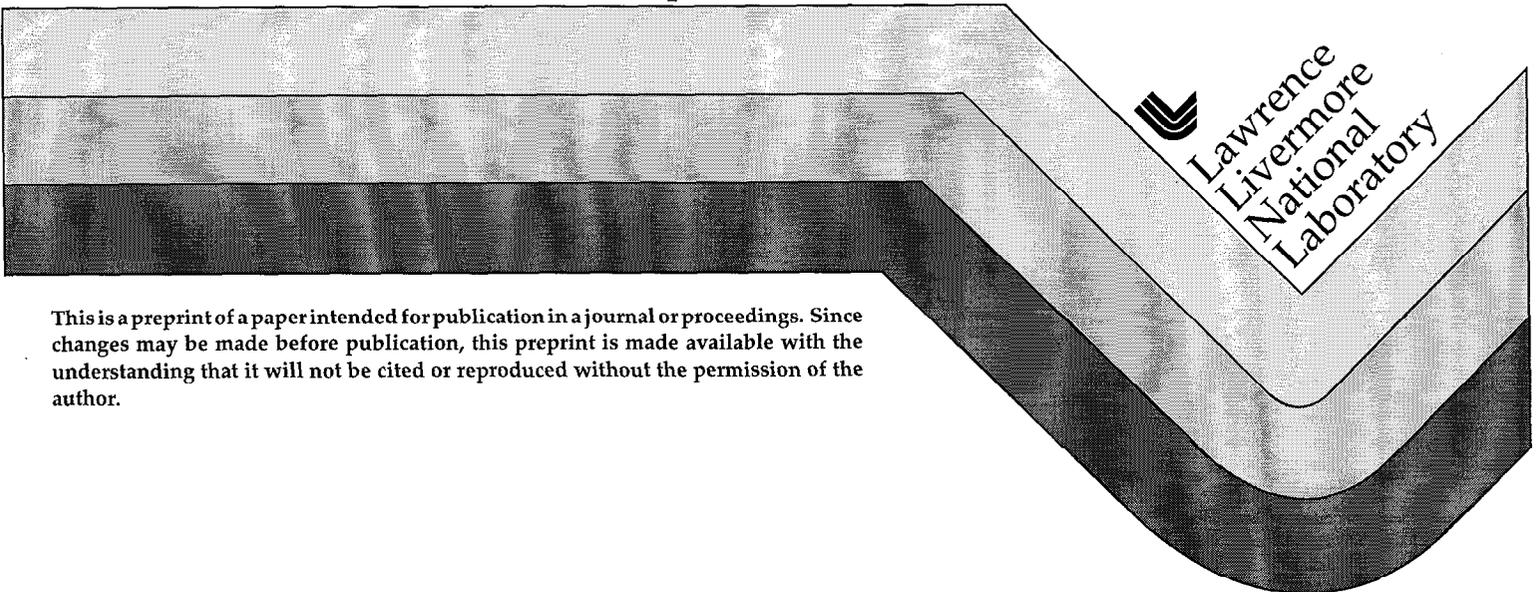


An Object-Oriented Approach for Development and Testing of Parallel Solution Algorithms for Nonlinear PDEs

R.D. Hornung
C.S. Woodward

This paper was prepared for submittal to the
Society for Industrial and Applied Mathematics
Workshop on Object-Oriented Methods for Inter-Operable
Scientific and Engineering Computing
Yorktown Heights, NY
October 21-23, 1998

September 17, 1998



This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

DISCLAIMER

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

An Object-Oriented Approach for Development and Testing of Parallel Solution Algorithms for Nonlinear PDEs *

Richard D. Hornung [†] Carol S. Woodward [†]

Abstract

An object-oriented design that provides flexibility in simulation codes is presented. This flexibility allows programmers freedom to easily change solution algorithms and discretization schemes as well as add new solver packages as they become available. Careful attention is paid to separating algorithm, data, and specific problem classes to provide for ease in changing any of these components. Furthermore, data structures are chosen so that each component works with data in a form best suited to its needs. Lastly, we present some experiences and comments on the tradeoffs involved with this design.

1 Introduction

We present an object-oriented software design for solving coupled systems of nonlinear, time-dependent partial differential equations (PDEs). We concentrate on implicit time solution methods for which there may be a variety of choices in components of the overall solution algorithm. This design targets algorithmic flexibility and extensibility which allow exploration of solution method alternatives. In particular, we separate solution methods, data structures and discretization schemes. Our goal is to build a framework that facilitates the exploration of algorithmic choices that arise when solving systems of nonlinear PDEs. However, this paper focuses on software design issues that have arisen during this development, especially those concerns related to the use of independently-developed solver technology.

Flexibility among solution algorithm components is often an important ingredient in the development of efficient, parallel simulation codes. In many cases, the “best” solution approach is often a pursuit of ongoing research, especially for large-scale, three-dimensional scientific applications. Codes may evolve substantially throughout their lifetimes forcing significant changes to numerical solution algorithms and discretization methods. Reasons for major changes may include advances in both linear and nonlinear solver technology, the extension of modeling capabilities such as, new physics and chemistry “modules”, or changes to PDE discretizations. Consequently, software flexibility and extensibility may be critical to useful application code development. Flexibility enables application code builders to examine various algorithmic choices and to incorporate software packages developed apart from the application project. Extensibility allows individual software components to be

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48.

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, <http://www.llnl.gov/CASC/>, hornung1@llnl.gov, cswoodward@llnl.gov

changed with minimal impact to others. These software characteristics can be achieved by imposing an appropriately loose coupling between individual components within the code.

In the discussion that follows, we describe an object-oriented approach that achieves solution component decoupling. Section 2 presents an example problem which we use as a reference point to motivate and illustrate our approach. In Section 3, we overview our class and data structure design. Lastly, in Section 4, we summarize our experiences and make some recommendations for the design of flexible application codes.

2 Example Problem

We consider, as an example, the nonlinear, time-dependent PDE system consisting of two equations and two unknowns given by,

$$(1) \quad \frac{\partial s(p)}{\partial t} - \nabla \cdot K_s(s, p) \nabla p = q_s,$$

$$(2) \quad \frac{f(s, p) \partial p}{\partial t} - \nabla \cdot K_p(s, p) \nabla p = 0,$$

where s and p are the primary variables, and t is time. The functions K_s, K_p , and f represent nonlinear coupling between the variables and q_s is a function of only the spatial position.

There exist a variety of options for generating an approximate solution to the system (1)-(2). Alternatives involve choices for spatial discretization, discrete time integration, and solution of nonlinear equations. For instance, assume we have applied a spatial discretization scheme to (1)-(2) and an implicit discretization in time. We then arrive at a set of coupled discrete nonlinear equations that must be solved to advance the unknowns through a sequence of discrete time increments. Given this scenario, there are two main alternatives for advancing the system in time. One may treat the equations using a single nonlinear solve that advances both variables simultaneously. Or, the system of equations may be decoupled and integrated in parts using an appropriate sequential time-stepping approach. In either case, it is often unclear how best to choose time increments. These increments may increase or decrease due to accuracy considerations or the convergence behavior of nonlinear iteration processes.

For either of the above time advance alternatives, one or more nonlinear systems must be solved at each timestep. Each nonlinear system can be solved using one of many formulations and extensions of Newton's method (inexact methods, globalized methods, etc.), or another method such as nonlinear multigrid. If a variant of Newton's method is adopted, linear Jacobian systems are solved at each iteration of the nonlinear solution process. There are many choices for solving and preconditioning these linear systems as well. Choosing appropriate linear and nonlinear solver technology, especially a good preconditioner can be difficult.

The above choices motivate the need for flexibility in scientific codes. Furthermore, we note that a number of software packages are available which solve subproblems found in discretizing equations similar to (1)-(2). For instance, the KINSOL package [6], developed in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL), provides robust nonlinear solver technology using an inexact Newton-Krylov method with a line-search globalization procedure [2] and a GMRES iterative linear solver [5]. The PETSc package, developed at Argonne National Laboratory also provides robust nonlinear solver technology using similar algorithms [1]. Moreover, the PETSc package provides linear solvers of various forms as well as preconditioners. The HYPRE

package, also under development in CASC at LLNL [3] will provide a suite of robust, parallel linear solver preconditioners for a wide variety of linear algebra problems. Extensibility in a code would allow use of these packages to compare and contrast the implemented algorithms. To benefit from code flexibility and extensibility, we have developed an object-oriented design that will allow easy changes to algorithms and use of packages.

3 Framework Design

The essential components of a nonlinear solution algorithm include a nonlinear solver and a linear solver which may include preconditioning. We want to easily change nonlinear solvers and linear solver preconditioners without impacting other parts of the application code. We also want to change the way discrete quantities are used in the approximation without having these changes affecting the way we use the solvers. Thus, our framework design focuses on the appropriate encapsulation of these independent functional parts and flexible use of data structures among them. In this section, we describe an object-oriented framework that provides us with flexibility to explore algorithmic options and inter-operability.

The class organization in our framework is based on the model for application development provided by SAMRAI [4], an object-oriented C++ software framework for structured adaptive mesh refinement (AMR) applications and under continuing development in CASC at LLNL. SAMRAI provides computational scientists with fundamental AMR support as well as other general, extensible software components useful in parallel structured mesh computations. For example, SAMRAI manages the computational mesh, simulation data, and parallelism for an application. SAMRAI was adopted for our work because it is useful for developing algorithmic framework structure, and we plan to eventually explore the combination of nonlinear solution methods and adaptive mesh refinement.

Since the essential classes in our framework are built using components of SAMRAI, we mention briefly the relationship between application data management and the computational mesh employed in that infrastructure. In conventional structured AMR methodology, computational cells corresponding to a single level of spatial mesh refinement are clustered to form a collection of *patches*. Typically, simulation data lives on each patch as a set of logically-rectangular arrays. Using SAMRAI variable representation and data management functionality, we implement numerical routines over single patches. The separation of the nonlinear problem discretization from details of data storage on the mesh is important for two reasons. First, the patches and their data may be distributed across a parallel machine without affecting the implementation of numerical kernels. Second, the numerical routines do not need to change if different solvers with different storage schemes are used.

3.1 Overview of Class Structure

Our framework design centers on six primary classes that are used to orchestrate the nonlinear solution process: `NonlinearTimeIntegrator`, `NonlinearSolverAlgorithm`, `NonlinearSolver`, `LinearSolver`, `NonlinearPatchModel`, and `Vector`. Next, we outline the organization and functionality of the first five classes. The `Vector` class is a key link between the data management features in SAMRAI and the use of different solvers and will be discussed in Section 3.2.

An illustration of the relationships between these classes and some of their basic functionality is shown in Figure 1. The `TimeIntegrator` class manages overall problem initialization and the time integration process. It is responsible for coordinating the

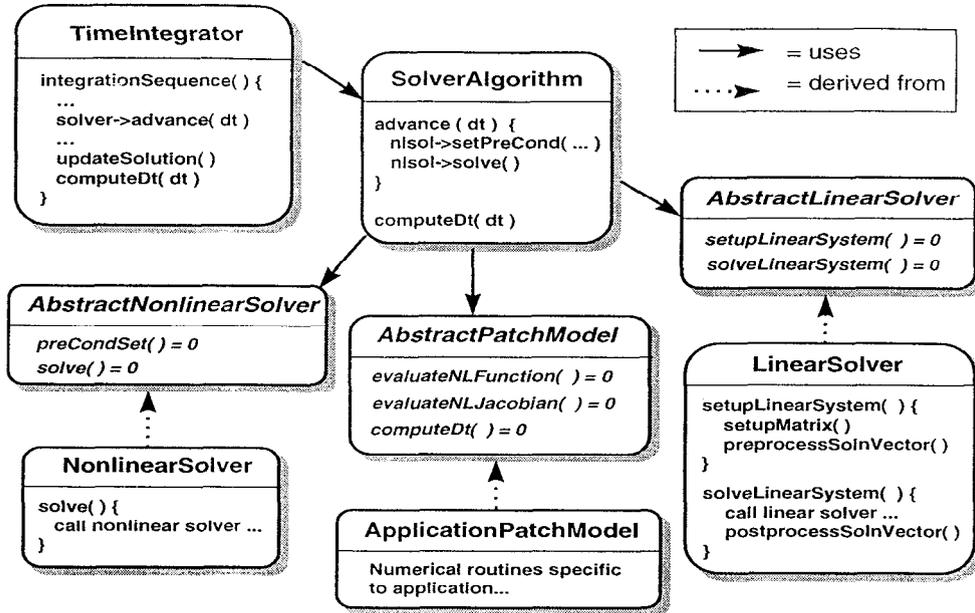


FIG. 1. The configuration of primary algorithmic classes in the nonlinear framework. Abstract classes are represented with italics, concrete classes are represented with regular type.

various nonlinear solves that are used to advance the data for the discrete time-dependent PDE problem. Each nonlinear solve is modeled by a SolverAlgorithm object. The SolverAlgorithm object coordinates the actions between the NonlinearSolver object, the LinearSolver object, and the ApplicationPatchModel object. The NonlinearSolver provides the solver for the nonlinear system, the LinearSolver provides the preconditioner for the Jacobian system, and the ApplicationPatchModel provides patch-based numerical routines for the discrete problem, such as calculation of the Jacobian matrix entries or the nonlinear residual.

The TimeIntegrator class coordinates nonlinear solves depending on the discrete time-stepping sequence. If the problem is treated as a single fully-coupled nonlinear system of discrete equations, TimeIntegrator manages the sequence of discrete time steps and invokes a single nonlinear solve to advance the data through each increment. If a sequential approach is used, TimeIntegrator orchestrates the discrete timestepping process involving different solvers and controls the synchronization of data among them via the “updateSolution” function. The basic operations defined by the integrator class involve problem setup and initialization, and problem solution.

The TimeIntegrator class can be applied to any nonlinear problem that is integrated in an implicit fashion. Several SolverAlgorithm objects can be registered with it. A unique solver algorithm object is needed for each distinct nonlinear solution process used in the overall time integration. For example, a single, fully-coupled nonlinear system treatment requires a single SolverAlgorithm object, whereas several different SolverAlgorithm objects are coordinated when a sequential approach is used. The actual integration sequence used depends on the order in which the solver algorithms are registered and the timestep information that they provide to the integrator when queried for this information via the “computeDt” function. If more general algorithmic functionality is needed, the integrator

advance routine may be specialized through class derivation to allow for arbitrarily complex time integration sequences.

The `SolverAlgorithm` class encapsulates the components and operations needed for a single iterative nonlinear solution process. This class coordinates objects that define the discrete problem, the nonlinear solver, and the Jacobian linear system preconditioner. However, all details of the interaction between these entities are hidden from the time integration manager. The interface that the solver algorithm exposes to the time integrator includes generic function calls for initializing and invoking the nonlinear solution process and for providing time increments. To perform the operations required to solve a nonlinear problem, a `SolverAlgorithm` object is initialized with objects defined by concrete subclasses of `AbstractNonlinearSolver`, `AbstractLinearSolver`, and `AbstractPatchModel`.

The `AbstractNonlinearSolver`, `AbstractLinearSolver`, and `AbstractPatchModel` classes are abstract base classes that define the interfaces between the solver algorithm and the specific implementations of these parts of the solution procedure. In other words, the `NonlinearSolver` base class defines the interaction between the solver algorithm object and the nonlinear solver, the `LinearSolver` class defines the interface to the Jacobian system preconditioner, and the `ApplicationPatchModel` defines the calls to the routines that implement the discrete equations, i.e. nonlinear residual and Jacobian matrix calculation.

A concrete nonlinear solver class is derived from the `AbstractNonlinearSolver` base class and serves to “wrap” function calls and data structures specific to a particular nonlinear solver package. Thus, a `SolverAlgorithm` object may initialize a nonlinear solver’s parameters, marry it with a preconditioner (“`setPreCond`” function), provide it with a nonlinear residual calculation routine via the patch model object, manipulate the data in its solution vector, and use it to solve a nonlinear problem without knowing any details of the solver implementation. The only caveat is that the nonlinear solver be implemented so that it can use the vector structure that we provide in the framework (see Section 3.2). KINSOL and PETSc are two examples of solver packages have been developed in this way.

The `AbstractLinearSolver` class in Figure 1 serves to define the interface between the `SolverAlgorithm` and the concrete `LinearSolver` in a similar fashion. The `LinearSolver` subclass provides the preconditioner for the Jacobian systems generated by the nonlinear solver object. However, unlike nonlinear solvers, most linear solver and preconditioner libraries store linear system data and generate solutions using data structures for matrices and vectors that are specific to the needs of the solution process. The “`setupMatrix`”, “`preprocessSolnVector`”, and “`postprocessSolnVector`” functions are implemented in the concrete `LinearSolver` class to manage storage between the linear solver on the one hand and the nonlinear solvers and numerical kernels, on the other. These data storage issues will be discussed further in Section 3.2.

Finally, the `ApplicationPatchModel` class is derived from the `AbstractPatchModel` base class. It provides numerical routines that define the specific discrete problem under consideration. That is, the patch model is used to compute the nonlinear residual, Jacobian matrix entries, and boundary conditions for each patch on the mesh. The details of these routines are hidden from the nonlinear solver and preconditioner. The solver objects are supplied with the correct function calls implemented in `ApplicationPatchModel` when the `SolverAlgorithm` object initializes them.

In our framework, numerical operations use the language of SAMRAI “variables”. Variable quantities used in the discrete approximation are registered with the `SolverAlgorithm` class by its `ApplicationPatchModel` according to the role the variables will play dur-

ing the solution process. Typically, a nonlinear solution quantity is registered as “NL_SOLN_TIME_DEP”. A static quantity that does not change in time as the problem evolves but which needs to be stored is registered as “INPUT”. Lastly, a temporary quantity that is only used for scratch computations is registered as “TEMPORARY”. If the example problem in Section 2 is solved as a coupled system using a single nonlinear solve, both s and p are “NL_SOLN_TIME_DEP” variables and q_s is an “INPUT” variable. The quantity K_i may be treated as a “TEMPORARY” since it may be computed separately but need not be stored between calls to evaluate the residual or Jacobian. The `SolverAlgorithm` class manages the storage for each variable in a manner consistent with the way the variable is registered. Thus, a link is established between the patch model and the solver algorithm whereby the model specifies the data that it needs and the algorithm manages storage for this data. The fact that the problem data is further translated into the language of vectors for the linear and nonlinear solvers is unknown to the patch model routines.

3.2 Data Structures

Data structures for large-scale application codes should be chosen so that memory access characteristics are appropriate for the operations required by the computational methods employed. However, it is generally impossible to develop data structures that provide minimal memory access time for all operations involved in a complicated scientific computing application. Thus, we have chosen to let the needs of solution algorithm components dictate data structure choice so that data for the nonlinear solver and the Jacobian system preconditioner are managed independently for performance reasons. In this section, we present the `Vector` class that is used to coordinate the interaction between solvers maintained by `SolverAlgorithm` in Figure 1.

3.2.1 The Vector Class Recall that SAMRAI manages patch distribution and data communication within our solver framework. The `Vector` class is built from SAMRAI components and supplies vector kernel operations to the nonlinear solver. Since application-specific numerical routines, such as the nonlinear residual calculation, are integral to the nonlinear solution process, these numerical routines access the same data storage as the vector kernel. That is, the `Vector` class encapsulates the concept of a vector that can be understood by `SolverAlgorithm`, `NonlinearSolver`, and `ApplicationPatchModel`. The basic structure of the `Vector` class is illustrated in Figure 2.

The `Vector` class defines a vector to be a collection of data objects distributed across the patches forming a computational mesh. Recall that in SAMRAI data is managed on a computational mesh composed of “patches”. On a patch, data is stored as an array of data components, each of which represent a variable quantity in the application. A `Vector` object is composed of a subset of these patch data components over a collection of patches. Vector kernel operations are provided for each vector component and depend on the data type of the component, such as cell-centered or node-centered. In the example system, (1)-(2), the solution vector maintains two data components that correspond to the unknowns s and p . If these variables are represented on the mesh as cell-centered quantities, then this information is known to the vector which provides kernel operations for each component. Our vector class allows any combination of distinct mesh quantities that may be cell-, face-, or node-centered on the mesh. The main point here is that a `Vector` contains information about the mesh structure and the form of data on the mesh.

The variable registration process described in Section 3.1 establishes the link between

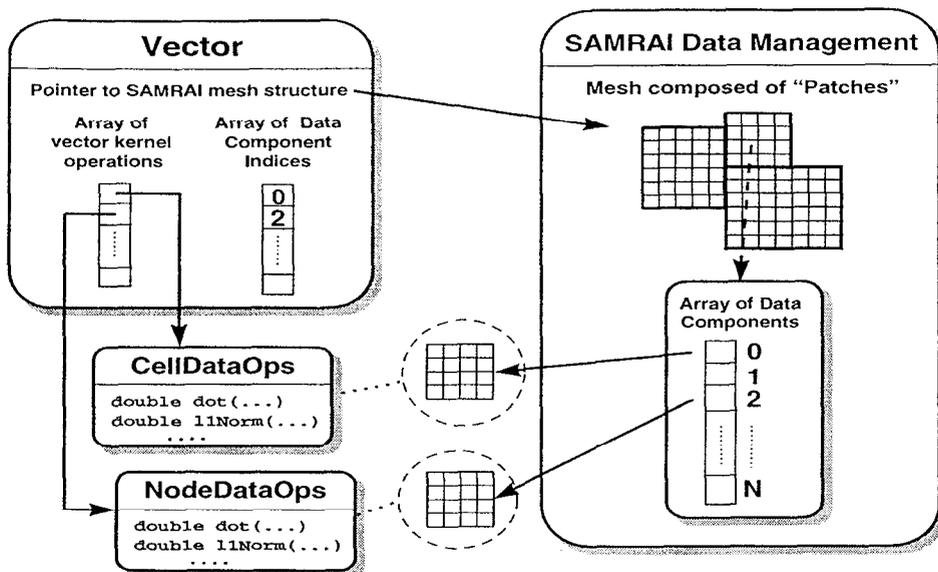


FIG. 2. A vector with application variables 0 (cell-centered) and 2 (node-centered).

data in the patch model and data in the vector. The patch model registers its variables with the solver algorithm which adds the patch data component information to the vector. Once the registration process is complete, the nonlinear solver can clone and destroy vector objects as needed, and the patch model can perform its operations independent of the vectors. This process causes no conflicts since data is moved into separate storage to supply numerical routines with ghost cell information when they involve stencil operations on the mesh. Ghost cell space is used when vector data is needed in stencil routines, but is not used for basic vector kernel operations. Thus, basic vector calculus operations, such as dot products, norms, and vector additions (used extensively during the nonlinear solution process) execute using loops with stride length 1 for efficient performance. All cloned instances of a vector share the same ghost cell storage through an “arena” mechanism. This sharing minimizes storage requirements and allows the transparent communication of data between nonlinear solution algorithms and discretization routines.

3.2.2 Linear Solver Data In our framework, the solvers and application code are independent. Thus, a preconditioner package, such as HYPRE, is allowed to manage storage in a manner suitable for the efficient parallel construction of solutions to linear systems. Referring to the illustration in Figure 1, the Jacobian matrix coefficients are computed in `ApplicationPatchModel` using variable quantities that describe the discrete PDE problem. The concrete `LinearSolver` object is responsible for translating the linear system into structures associated with the linear solver package. However, we emphasize that no more than one copy of the matrix coefficients is maintained. The preconditioner solve solution and right-hand-side vectors are translated between storage schemes as well.

Maintaining separate storage for the linear and nonlinear systems is reasonable for several reasons. First, the application code can manipulate solution data in terms of quantities in the discrete numerical problem, such as when evaluating the nonlinear residual.

Second, the solution of linear systems is usually the most costly part of many computations and so the solution procedure should be as efficient as possible. Once the linear system matrix problem is specified, the linear solver does not need further details about the discrete problem. Third, solver components may be interchanged with minimal impact on each other or on the application code, and solvers can be developed and enhanced independently.

3.3 Languages

In our framework, we combine C++ class descriptions for the high level parallel framework with FORTRAN 77 computational kernels. The solver packages are written in C. We believe these language choices are expedient for our purposes. Since we are concerned with structured meshes only, we are able to take advantage of this inherent structure. When numerical routines are needed in the nonlinear problem class, we are able to identify a start and end point for the computations. We thus send those to FORTRAN routines which can efficiently calculate the desired quantities.

4 Conclusions

We have presented a design for flexible and extensible scientific simulation codes. The two main ingredients of our design are the separation of algorithmic components and the careful choice of data structures. By encapsulating the solution algorithms, numerical routines and data structures, we have reduced the inter-dependence of function components which provides sufficient flexibility to easily switch nonlinear and linear solvers as well as discretization schemes. The main decision regarding data was to use structures for each component of the framework that were well-suited to that particular functionality. Specifically, we chose to copy vectors needed for the linear solver into the most appropriate structures for the solver rather than try to use a single vector structure throughout the code. In doing this copy, we have data in a structure that allows greater efficiency for both the linear and nonlinear solvers. This structure, defined in the Vector class, allows stride 1 access to elements for fast kernel operations (axpys, dot products, norm calculations, etc.). These actions are entirely transparent to the patch model routines which define their operations in terms of the problem variables.

The design presented in this paper shows how a simulation code can be developed to model complex applications but still allow easy changes in algorithms, discretizations and data structures. With this flexibility, an application programmer can easily compare solution methods for complex problems. Lastly, we note that much of the software initially used in our framework is currently under development. Use of “new” packages is common in many large-scale scientific applications, and we note that our design provides extensibility to the applications programmer for easy accommodation of future changes.

References

- [1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11-Revision 2.0.22, Argonne National Laboratory, 1998.
- [2] P. Brown and Y. Saad, *Hybrid Krylov methods for nonlinear systems of equations*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 450–481.
- [3] E. Chow, A. Cleary, and R. Falgout, *Design of the HYPRE preconditioner library*, in SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, M. Henderson, C. Anderson, and S. Lyons, eds., Philadelphia, PA, 1998, SIAM.

- [4] X. Garaizar, R. Hornung, and S. Kohn, *The use of object oriented design patterns in the SAMRAI structured AMR framework*, in SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, M. Henderson, C. Anderson, and S. Lyons, eds., Philadelphia, PA, 1998, SIAM.
- [5] Y. Saad and M. Schultz, *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856–869.
- [6] A. G. Taylor and A. C. Hindmarsh, *User documentation for KINSOL, a nonlinear solver for sequential and parallel computers*, Tech. Rep. UCRL-ID-131185, Lawrence Livermore National Laboratory, 1998.