Development of Generic Field Classes for Finite Element and Finite Difference Problems

DIANE A. VERNER[1], GREGORY L. HEILEMAN[1], KENT G. BUDGE[2], AND ALLEN C. ROBINSON[2]

[1]*Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131*

[2]*Computational Physics Research and Development (1431), Sandia National Laboratories, Albuquerque, NM 87185-5800*

ABSTRACT

This article considers the development of a reusable object-oriented array library, as well as the use of this library in the construction of finite difference and finite element codes. The classes in this array library are also generic enough to be used to construct other classes specific to finite difference and finite element methods. We demonstrate the usefulness of this library by inserting it into two existing object-oriented scientific codes developed at Sandia National Laboratories. One of these codes is based on finite difference methods, whereas the other is based on finite element methods. Previously, these codes were separately maintained across a variety of sequential and parallel computing platforms. The use of object-oriented programming allows both codes to make use of common base classes. This offers a number of advantages related to optimization and portability. Optimization efforts, particularly important in large scientific codes, can be focused on a single library. Furthermore, by encapsulating machine dependencies within this library, the optimization of both codes on different architectures will only involve modification to a single library. © 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

This research addresses the development of a general-purpose object-oriented class library for use in problems requiring operations across large arrays. Potential applications include finite difference or finite element approximations to differential equations as well as general matrix libraries. Large computational physics problems often make use of either finite difference or finite element methods. Although these numerical techniques are quite different, they do share a similar set of underlying array computations. Furthermore, increasing the computational speed of the software developed for these methods has, until recently, focused on the use of vector supercomputers. The availability of a variety of powerful massively parallel computers must now also be considered in developing code. Thus, portability was one of the primary concerns addressed during the development of the array library. This array library takes advantage of the ease of access offered by workstations, as well as the high performance offered by vector and massively parallel architectures.

The porting of parallel codes to different platforms is often a prohibitive task because it typically involves rethinking a problem entirely in or-

Received April 1993
Revised June 1993

This research was performed in part at the University of New Mexico under a grant from Sandia National Laboratories (contract number AE-1518) and in part at Sandia National Laboratories supported by the U.S. Department of Energy under contract number DE-AC04-76DP00789.

Scientific Programming, Vol. 2, pp. 227–234 (1993)
CCC 1058-9244/94/040227-08

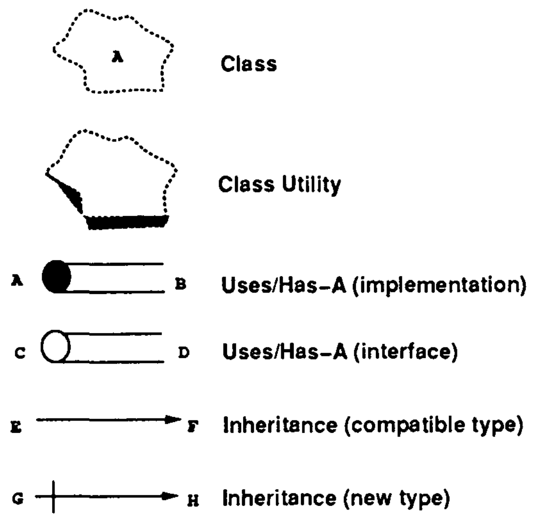


FIGURE 1 Relationships used in class diagrams.

der to take advantage of a specific computing platform or programming environment. Encapsulating machine-dependent portions of the code at the lowest class levels improves portability by limiting the number of classes that must be modified when porting to new architectures. To achieve these goals a distributed object-oriented design was used to develop this class library.

To capture both the functional and temporal characteristics of a software system at various levels of abstraction, a number of object-oriented diagramming techniques can be used. The structure of the generic classes developed here will be depicted using Booch's graphical object-oriented diagramming techniques [1]. Booch introduced several diagrams that can be used to represent these types of relationships. These include class, object, module, state transition, and process diagrams. The static structure of the classes developed in this article is represented using the class diagram relationships shown in Figure 1. A class utility, shown in Figure 1 as a shadowed blob, is defined as a collection of related subfunctions that are not contained within a class. In Figure 1, class A's implementation either uses one or more of class B's methods in its computations, or class A contains class B within its data members. The next relationship in Figure 1 shows class C's interface using the resources of class D. By contrast, class E inherits all of its attributes and methods from class F. The addition of the line perpendicular to the inheritance line indicates that derived class G is not type compatible with base class H.

In Section 2, we describe the hydrocode applications that use the array library. Although these applications motivated the development of the array library, they are by no means the only applications that can make use of this package. Section 3 discusses the design of classes in the array library, demonstrates their use in two existing scientific codes, and then compares them to the original field classes used in one of these codes.

2 HYDROCODES

Simulation codes that model phenomena such as the impact of solid bodies at high velocities or the effect of high explosive detonation are commonly termed hydrocodes [2]. Sandia National Laboratories has developed two hydrocode packages called PCTH and RHALE++. PCTH is based on a finite difference method, whereas RHALE++ is based on a finite element method. Because the class library discussed here was developed to support these existing codes, this section will briefly consider these codes and point out their differences.

2.1 Existing Hydrocodes

Finite difference methods involve converting partial differential equations into algebraic equations. PCTH is a parallel hydrocode based on a finite difference methodology to approximate the equations of mass, momentum, and energy conservation. PCTH is based on a three-dimensional fixed-mesh finite difference method in which the material flows through the mesh. A two-step explicit solution scheme is used to integrate the equations of motion forward in time. The first step is a Lagrangian step in which material motion is calculated. Conservation of mass, momentum, and energy must be satisfied across this step. The second step is a remap step in which the Lagrangian state quantities are mapped back to the fixed mesh. This two-step approach makes it easier to handle multiple materials in the simulation [3].

Finite element methods are also used to approximate the solution of partial differential equations. In finite element methods, each region of interest is subdivided into finite sets of elements connected together at a set of points called nodes. In these methods, the solution is defined everywhere using a piecewise-polynomial approximation [4]. RHALE++ is a hydrocode that uses a finite element method coupled with an arbitrary Lagrangian-Eulerian mesh motion. RHALE++

solves the equations of motion on an unstructured finite element mesh. This approach allows the simulation of odd-shaped structures.

Sequential codes for shock wave physics simulation have been separately developed for both of these numerical approaches using the object-oriented programming language C++. These codes are currently in use at Sandia National Laboratories [5]. Although these two numerical approaches are quite different, software modules that are capable of representing scalar, vector, and tensor fields are fundamental components in both approaches. Our design captures the commonality of these fields and encapsulates machine dependencies. The advantage offered by this approach is that the implementations of both numerical techniques can share a common set of underlying software classes, and the specific numerical class libraries can then be built on top of these base classes through the use of inheritance. The time spent on future low-level efficiency improvements and machine optimizations can then be shared immediately by all derived classes and thus all codes (PCTH and RHALE++) that use these base class libraries. This approach greatly simplifies the porting of both codes to different computers because only a single library of base classes needs to be modified.

2.2 Parallel Object-Oriented Hydrocode Simulations

Hydrocode simulations are typically compute intensive and also require large amounts of memory. To decrease execution time, these programs are often run on parallel or vector machines. For parallel computers, memory and processing time are divided among the processors. The simulation must be designed in such a manner as to allow parallel processing. Hydrocodes are well suited for the data parallel programming model because their algorithms typically make use of large arrays.

Parallelism in the data parallel model is exploited by performing simultaneous operations across large sets of data, rather than by having multiple threads of control [6]. For example, a single program statement may simultaneously add all of the elements of two large data sets. This style of programming is well suited for fine-grained single instruction multiple data (SIMD) machines. In a SIMD computer, the processors operate in lockstep using a global clock. In addition, data parallel algorithms have been successfully used on medium-grained multiple instruction multiple data (MIMD machines). Data parallel algorithms intended for MIMD computers are often referred to as single program multiple data (SPMD) [7]. On an MIMD computer, data parallel operations are performed by synchronizing all processors after each step or several small steps. Synchronization and communication must be explicitly performed by the programmer but overall communication costs decrease because several operations may be executed between communication events.

PCTH is implemented using an SPMD coding style. Using this approach, PCTH decomposes the problem domain into rectangular blocks. Additional cells on each block edge, called ghost cells, are created to allow data from neighboring blocks to be stored. As shown in Figure 2, these blocks are called hydroblocks. Each processor is allocated one or more hydroblocks. Each hydroblock contains vector, scalar field, and vector field objects. Single dimensional quantities such as initial momentum and gravity are stored as VECTORs. Multidimensional values such as cell temperature, volume, and pressure are stored as cell-centered fields in a class called CC_FIELD. In a cell-centered field, the grid is fixed in space and the values at the center of each cell are mapped to the indices of the array variables. In contrast, in a face-centered field the values on the face of each cell are mapped to the indices of the arrays. Velocity is stored as an instance of the class FCV_FIELD (face-centered vector field), which is an array or vector of face-centered fields. The state of the materials used in the simulation is contained in objects of the class MATERIAL. Each hydroblock object also contains information concerning required output. When RHALE++ is parallelized a similar SPMD programming style will be used.

3 GENERIC FIELD CLASS DESIGN

As stated previously, large physics simulations rely heavily on the use of a basic set of mathematical operations. At the lowest levels of calculations, element-by-element operations are performed on ordered sets of values. No topology or calculus is included at this level. These operations are encapsulated into a FIELD class for full precision floating point numbers and an IFIELD class for integer data elements. It should be noted that the usage of field as our class name is not strictly correct, because the concept of a field implies topology and calculus, and our generic classes have neither. However, the names ORDERED_SE-

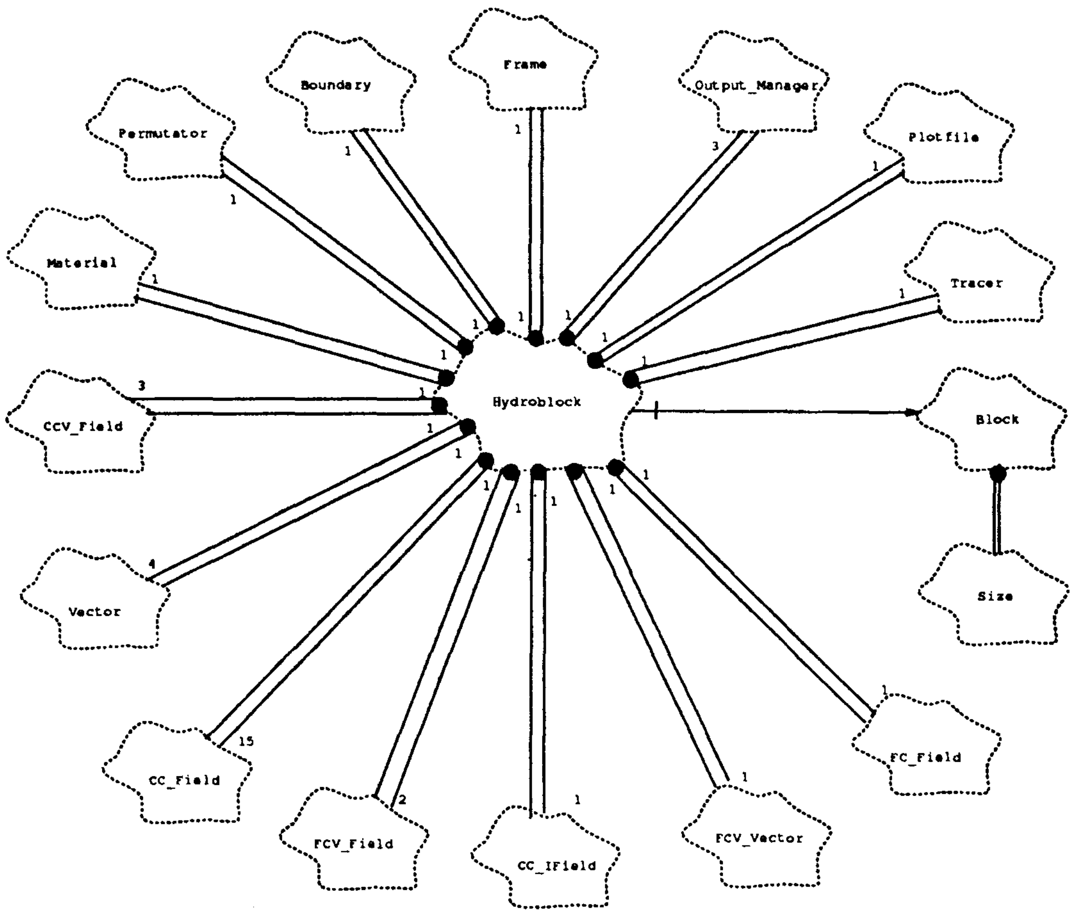


FIGURE 2 PCTH hydroblock class structure.

T_OF_DOUBLE and ORDERED_SET_OF_INT are awkward and the use of ARRAY seemed likely to lead to name clashes with existing programs. In this article, the terms FIELD and IFIELD will refer specifically to the individual classes. We will also use the term field to refer generically to both the FIELD and IFIELD classes.

In developing scalar, vector, and tensor field classes, there are two approaches for selecting the structure of the data: arrays of objects or objects of arrays. In the arrays of objects approach the vector or tensor object is laid out sequentially in memory whereas in the objects of arrays approach each of the components of the vector or tensor objects is laid out sequentially in memory. In this approach, an array of data elements (or a pointer to an array) is selected as the object attribute. RHALE++ originally used an array of objects approach but difficulties in porting and optimizing became apparent. PCTH always utilized the object of arrays approach in order to leave open the possibility of porting to SIMD architectures. Therefore, the objects of arrays approach was adopted for RHALE++, which led to the development of a new field class. This field class has now been adopted by the PCTH project.

3.1 Original PCTH Field Class

Figure 3 shows the class diagram for the original PCTH field class library. In this library, the field classes contain all the mathematical operations and memory management functions required for their respective classes. The FIELD class contains a dynamic array of floating point data elements, an integer pointer to a reference counter, and a pointer to the topology class SIZE. The IFIELD class uses a dynamic integer array instead of a floating point array. The SIZE class contains data that specifies the dimension of the field class and the size of the array in each axis. Both field classes have a HAS-A relationship with the SIZE class. That is, one of the members of the class is a pointer to a SIZE object. A USES-A relationship

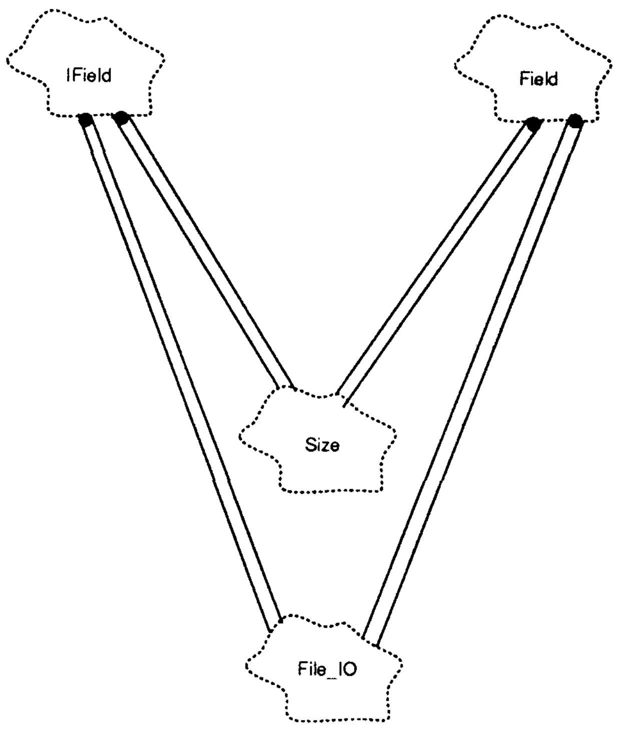


FIGURE 3 Original PCTH field class library class diagram.

exists between the field classes and the FILE_IO class. The field classes both call FILE_IO member functions when it is necessary to perform file-based I/O.

3.2 New Field Class

Figure 4 depicts the new field class library structure developed for RHALE++ and PCTH. As with PCTH, the floating point FIELD and integer IFIELD classes are used as the base classes for classes that contain topological information. These field classes have a HAS-A relationship with either the SNL_FIELDNODE class or the SNL_IFIELDNODE class, which are discussed in more detail below. That is, one of the data members in the field classes is a pointer to the appropriate SNL_FIELDNODE or SNL_IFIELDNODE class. The term *node* will be used to generically refer to the SNL_FIELDNODE and SNL_IFIELDNODE classes. The data members of the node classes consist of a reference counter, a length value, and a dynamic array of either integer or floating point numbers.

For efficiency reasons, the field class uses dynamically allocated memory and the reference counting memory management technique in order to decrease the number of heap accesses. This is especially important on certain computers, such as the Cray, where heap accesses are very expensive. This leads to the use of an envelope/letter class idiom [8]. In this idiom, the envelope class handles all message requests from the external world. The letter class is completely encapsulated within the envelope class. In this project, the field class acts as the envelope class and the node class acts as the letter class. The advantage of this idiom is in the division of memory management and mathematical operations. Specifically, the envelope class handles all mathematical operations and reference counting. The letter class is responsible for memory management functions. These node classes then interface with the memory manager to perform dynamic memory allocation and deallocation.

As shown in Figure 4, the field classes have a USES-A relationship with two class utilities: GENFUN and FAST. The GENFUN class utility contains generic procedures that perform a few simple integer and floating point operations, as well as some file read and write operations. The FAST

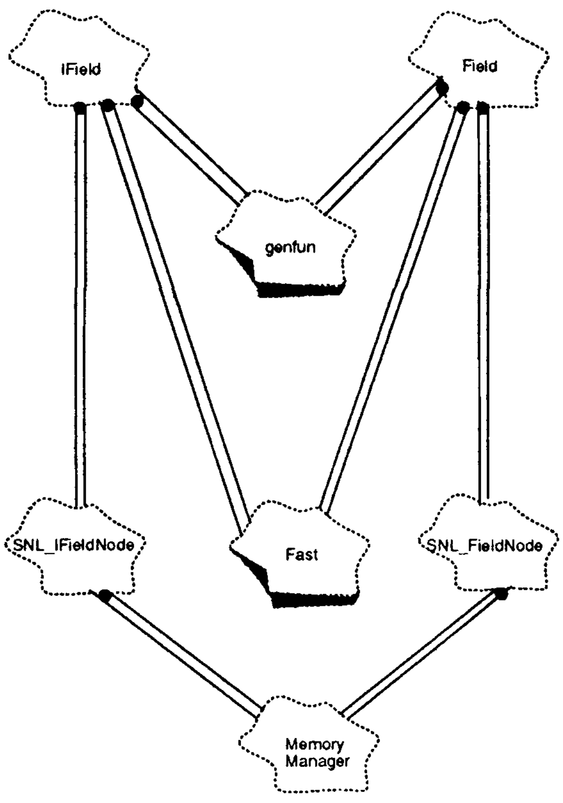


FIGURE 4 New generic field class library class diagram.

class utility contains vectorizable functions used to perform mathematical and logical operations on the field class members. This allows mathematical operations to be easily ported to many new machines and optimized without affecting the field classes. The field classes call the required vector functions from the appropriate overloaded operators or methods.

The new field class differs from the original PCTH class in several significant ways:

1. An envelope/letter class idiom is used. As discussed above, this leads to a better division between memory management and mathematical operations.
2. The field class contains no topological information about memory layout. This information is contained in derived classes, providing a more general and reusable set of base classes.
3. Memory allocation functions are contained in a separate class, thus allowing different memory management schemes to be implemented without affecting the field class design.
4. Vectorizable mathematical operations are contained in a separate class utility file that allows optimization of these operations in a different language, and also allows machine-dependent functions to be moved to a separate file from the field classes. The field classes then become machine independent.
5. A complete set of operations (not just those required by PCTH and RHALE++) have been implemented in these classes to enhance reusability in other applications.

In general, the PCTH field classes were simpler and easier to implement, but also more application specific and less complete, making them harder to reuse in other applications. Several important but potentially conflicting criteria were weighted when developing the new field classes: reusability, memory usage, execution time, and portability.

3.2.1 Reusability

The methods developed for the field class library are intended to be complete and extendable so that the library can be used in many different applications. Care was taken to create the minimum number of functions necessary, to avoid having an unmanageable class library. These classes perform array-scalar and array-array operations for both integers and floating point numbers. C++ allows the use of both functions and overloaded operators in its classes. There are 33 overloadable arithmetic operators defined in C++ [9]. Of these only the increment and decrement operators were not implemented because their functionality can be achieved using the binary plus and minus operators. The other functions required for the field classes can be divided into several categories: trigonometric, other transcendental functions, general purpose, and Fortran-like functions. There are also some specialized functions that either simulate hardware commands found on certain machines or are widely used in both finite element and matrix applications. All of the functions used by these field classes are detailed in [10]. Other application-specific functions are implemented in derived classes. For example in PCTH, the FC_FIELD and CC_FIELD classes, which are derived from fields, implement various mean, difference, and product functions required for this application. A mechanism to implement these additional functions has been provided through a protected method that returns a pointer to the start of the data element array.

3.2.2 Memory Usage

In general, hydrocodes use large amounts of memory that must be managed effectively to provide an efficient simulation. The type of computer being used to perform the simulation will affect the memory management requirements. Both the program executable size and the amount of program data used can affect performance. On computers with virtual memory capability, program data and executable segments may be swapped out to disk and large problems can (at least in principle) be run. However, the amount of dynamic memory used will significantly affect the speed of the simulation because more operations are being performed and more virtual memory accesses will occur. The processing nodes of many parallel computers have a fixed memory size that requires minimal executable size in order to maximize user memory for efficient scaling.

Dynamic memory allocation is performed by the field class and for efficiency reasons memory management is not relegated to the operating system. The memory manager functions, which are called by the field class, limit heap accesses by creating a free store pool of pointers to unused but allocated memory. Because these functions are not strictly part of the field class, they will not be discussed any further in this article. Refer to Verner [10] for a discussion of the memory manager. There are several techniques that can be used by the field classes to reduce the amount of dynamic memory used. Unfortunately, these require that the user be aware of what is happening in the field class in terms of memory in the overloaded operators. Overloaded operators cause the creation of several unnecessary temporary variables [5]. For example, the expression A = B + C * D creates four temporary variables, two per operation, when no memory management techniques are applied. There are two techniques that can be applied by the application programmer to decrease the number of temporary variables that exist at any particular time. The first technique involves rewriting the above expression as:

```
A = C * D
A +=B
```

This method eliminates two of the temporary variables, and if used in conjunction with reference counting will eliminate a third temporary. Unfortunately, this method forces an unnatural programming style. The application programmer must also force temporary objects to go out of scope so they can be deallocated. This can be done in the application code by adding scope delimiters "{ }" around code segments.

3.2.3 Minimizing Execution Time

There are many techniques used to improve time efficiency in C++ programs, such as inline functions, reference counting, and machine optimized code. Inline functions can be used to reduce function call overhead by expanding the inline function code at the location of the function call but generally at the expense of increased code size. Therefore, inline functions were only used on small functions that are called often. Array operations are optimized by writing vectorizable or assembly language functions. These techniques lead to machine and compiler dependent code, and are supported by the modular development used here.

Compared to the previous PCTH implementation, the new field classes have yielded similar execution times but a slightly larger executable size. The increased executable size is due to the larger number of functions that have been implemented in order to develop the complete set of base classes necessary for reusability. This penalty can be mitigated somewhat by splitting the field class and their FAST class utility functions into separately compiled files, and then creating a library that contains these files.

Reference counting is another method, used by the field classes, for minimizing unnecessary copying of data and thus greatly improving efficiency. A nice feature of this approach is that it is completely hidden from the user. Reference counting is a technique where several items point to one memory location rather than creating a new item for each reference [8].

3.2.4 Portability

As mentioned previously, one of the primary goals of PCTH and RHALE++ is ease of portability. Specifically, these simulations must be easily portable to many different architectures including single processor workstations and workstation networks, MIMD massively parallel processors, vector computers, and potentially even SIMD computers. Portability can be enhanced by isolating machine-dependent portions of the code from machine-independent portions. This is not to say that all the code will run efficiently on all computers. Instead, the subset of the code that benefits from optimization, such as the vector math libraries, is stored in routines at the lowest level in the hierarchy to hide the details behind underlying generic objects [2]. This machine-dependent code is compiled according to preprocessor statements that specify the specific code for that machine. This effectively separates the physics portions of the code from the architectural details. In the new generic field class, the FAST class utility provides this machine-dependent code. It is being ported to and optimized on the nCUBE2, Intel Gamma, and the Intel Paragon massively parallel computers, as well as the CRAY vector computers and SUN workstations. Future machine-specific optimization for these individual machines can focus on the FAST class utility.

4 CONCLUSIONS

In this research we have successfully developed a reusable field class structure that is being used in two different computational physics codes; one code uses a finite difference method whereas the other code uses a finite element method. This is of critical importance to the PCTH and RHALE++ development teams as all future development of the field libraries will be synergistic to both projects. Great care has been taken to make these classes extendable (to aid in adding new functionality), and easily portable to new architectures. It is also worth noting that a modification of these classes has been proposed for inclusion in the ANSI C++ standard library [11].

ACKNOWLEDGMENTS

We would like to extend our appreciation to James Peery for his technical efforts.

REFERENCES

[1] G. Booch, *Object Oriented Design: With Applications.* Redwood City, CA: Benjamin/Cummings, 1991.

[2] A. C. Robinson, A. Ames, H. E. Fang, D. Pavlakos, C. T. Vaughan, and P. Campbell, "Massively parallel computing, C++ and hydrocode algorithms," *Comput. Civil Eng.*, pp. 519–526, 1992.

[3] J. M. McGlaun and S. L. Thompson, "CTH: A three-dimensional shock wave physics code, *Int. J. Impact Eng.*, vol. 10, pp. 351–360, 1990.

[4] A. Davies, *The Finite Element Method: A First Approach.* New York: Oxford University Press, 1980.

[5] K. G. Budge, J. S. Peery, and A. C. Robinson, *USENIX C++ Technical Conference Proceedings.* Portland, Oregon: USENIX Association.

[6] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Communications ACM*, vol. 29, pp. 1170–1183, 1986.

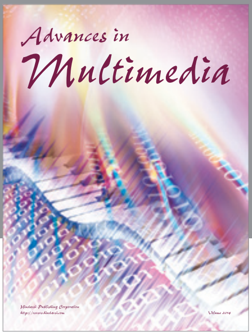
[7] M. J. Quinn and P. J. Hatcher, "Data-parallel programming on multicomputers," *IEEE Software*, vol. 7, pp. 69–76, 1990.

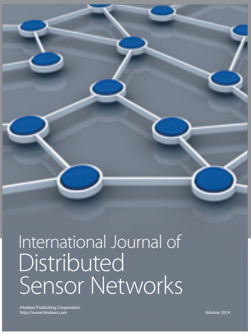
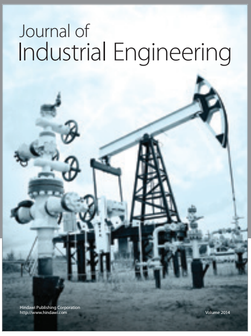
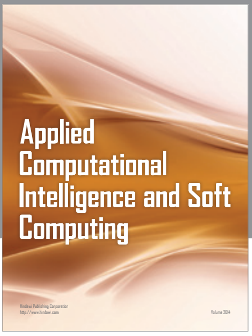
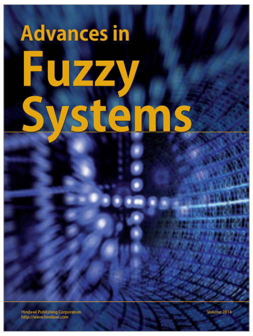
[8] J. Coplien, *Advanced C++ Programming Styles and Idioms.* Reading, MA: Addison-Wesley, 1992.

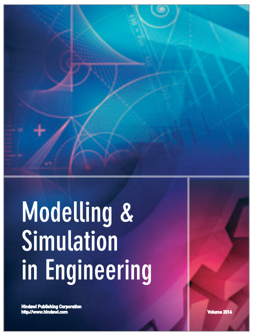
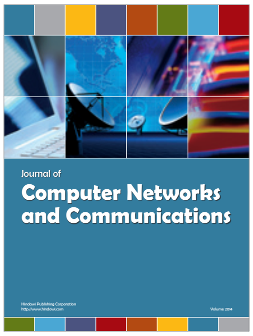
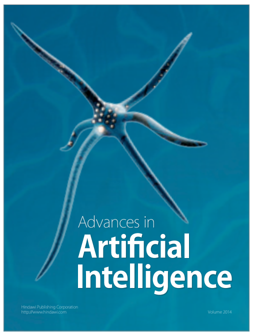
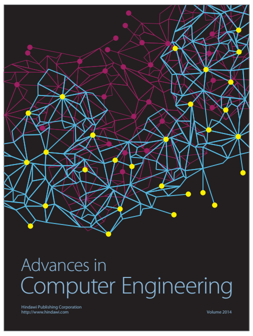
[9] B. Stroustrup, *The C++ Programming Language* (2nd ed.). Reading, MA: Addison-Wesley, 1991.

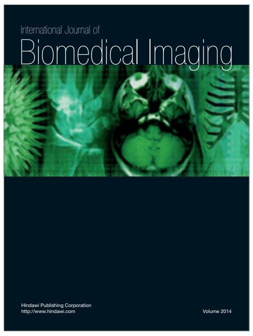
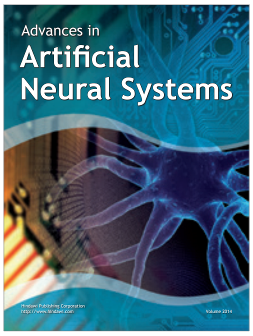
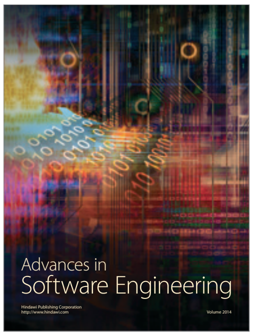
[10] D. Verner, "Developing generic classes for finite element and finite difference problems." Master's thesis, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, New Mexico, 1993.

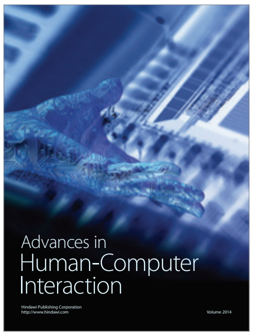
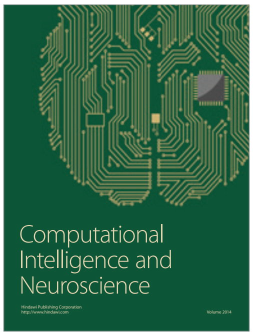
[11] K. G. Budge, *Proposal for a Numerical Array Library,* ANSI X3J16-93-0042/WG21-N0249, 1993.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems


Modelling & Simulation in Engineering

Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering