Object-Oriented Programming for Scientific Codes. II: Examples in C++

By T. J. Ross,[1] Member, ASCE, L. R. Wagner,[2] and G. F. Luger[3]

Abstract: This paper illustrates some specific numerical code applications using an object-oriented programming (OOP) paradigm. It is the second of two papers describing OOP issues for scientific-code development. A brief description is given of an object library as well as some simple finite difference and finite element user codes based on that library. The key to efficient OOP implementation in any environment is to conceive an effective object-oriented design of the physical problem and its mathematical abstraction in the first place. This design places a premium on understanding the physical problem from the perspective of class structure, inheritance, and communication. Some of the issues that must be addressed prior to the development of a truly successful OOP environment for large-scale scientific code development and employment are discussed. Optimization issues of OOP in general and the object-oriented programming language C++ in particular are discussed. Our research suggests that there is a tremendous potential for OOP as a development environment for scientific codes.

Introduction

Previous research efforts (Peskin and Russo 1988; Angus and Thompkins 1989; Forslund et al. 1990; Wagner et al. 1991; Filho and Devloo 1991) discussed the use of object-oriented programming (OOP) methods in the development of scientific codes for solving problems typically described by partial differential equations (PDEs). Implicit in these works was the desire to create an OOP environment where a large-scale scientific-user code is portable across a variety of architectures, where a user code is easily modifiable and maintainable, and where the solution process is efficient when compared to serial, vector, or parallel FORTRAN environments.

For scientific codes OOP provides distinctions among the continuum mathematics, the discretization of the mathematics, and the data structure. For any given architecture, the concurrency in the OOP approach is implemented internal to a small set of data structures (objects), and operations upon these objects, which will be generally applicable to a wide class of numerical procedures such as the finite difference and finite element methods. For emerging architectures involving massively parallel processors (MPP) the greatest hindrance to the full exploitation of the concurrency offered by distributed memory machines is that the programmer must explicitly specify the layout of the data storage for traditional languages like FORTRAN (Angus and Thompkins 1989). A major benefit of OOP is that the interface with a particular architecture can take place at the library-code level, where the machine dependent details of data storage, synchronization, and parallelism are implemented in lower-level objects, and not at the user-

[1]Assoc. Prof., Dept. of Civ. Engrg., Univ. of New Mexico, Albuquerque, NM 87131.

[2]Computer Sci., Kachina Technologies, Inc., Albuquerque, NM 87112.

[3]Prof., Dept. of Computer Sci., Univ. of New Mexico, Albuquerque, NM.

Note. Discussion open until March 1, 1993. Separate discussions should be submitted for the individual papers in this symposium. To extend the closing date one month, a written request must be filed with the ASCE Manager of Journals. The manuscript for this paper was submitted for review and possible publication on August 11, 1991. This paper is part of the ***Journal of Computing in Civil Engineering***, Vol. 6, No. 4, October, 1992. ©ASCE, ISSN 0887-3801/92/0004-0497/$1.00 + $.15 per page. Paper No.2381.

code level, which deals with the high-level objects associated with the mathematics of the physical problem.

The key to efficient OOP implementation in any environment, however, is to conceive an effective object-oriented design of the physical problem and its mathematical abstraction. This requires understanding the physical problem from the perspective of class structure, inheritance, and communication among classes. Angus and Thompkins (1989) propose to use data structures that mimic the formalism of the governing partial differential equations describing the mathematical abstraction of the physical problem. For example, in this way high-level objects would involve the continuum mathematics on the scalar and vector fields of the domain of the problem. Also at this level would be the operators, such as the divergence and the curl, that operate on the continuum fields. The next level of objects would be defined on the discretized continuum, where a particular scalar field on the continuum, for example, would become several special types of scalar fields on the discrete lattice. At this second level discrete forms of the operators and special classes of algorithms also would be specified, as well as boundary conditions. It is interesting that the specification of boundary conditions consumes most of the user code, but only a small amount (10%) of the computing time (Angus and Thompkins 1989). At this second level, and the highest level for that matter, there is no requirement as to how the discrete-field data structures are actually mapped onto the processors of a particular machine architecture.

The lowest level of object definition in this OOP hierarchy would implement the details peculiar to a particular hardware environment. And because these details are independent of the type of continuum specified by the PDEs, and because they are independent of the type of discretization employed, it seems prudent to encapsulate them within the lowest level objects in library code so that they can be reused for other applications. At this level the conversion operators need only know about the nature of the data storage (which is architecture-dependent) of the two discrete field types (scalar and vector) that they transform between (Angus and Thompkins 1989).

In terms of a particular OOP language implementation, Forslund et al. (1990), have done some very interesting work with a plasma particle simulation code written in C++. This plasma code, and numerous mass-transport problems such as those in fluid dynamics, have been addressed in the past using a variety of techniques like the particle-in-cell and vortex methods (Lu and Ross 1991), which lend themselves naturally to object decomposition. The code models the particles and fields (methods and data) of a two-dimensional (2-D) problem as a quadtree region and of a three-dimensional (3-D) problem as an octtree region in local coordinates. Although in FORTRAN the governing PDEs are solved by block-diagonal linear-algebraic methods, the C++ approach used a local iterative solve on the regions, then the continuum was solved through communication across boundary layers between the regions. For example, a class "particle" contains methods for advancing plasma particles in response to the electromagnetic fields, and "region" restricts the particles and fields to a particular space in the continuum. A class "neighbor" represents the overlap between adjoining regions and provides the basic communication buffering between the regions. A "region" would reside on a single processor in a parallel environment. As particles move around the discretized grid, they pass from one region to another, and the fields communicate across the boundaries of each region (Forslund et al. 1990).

Ideally, objects and the operations on them could be defined in the most general mathematical form, and the libraries could determine what specific operation is performed and even what specific discretization methodology is optimal. Since it is difficult to automate the selection of a discretization scheme, the user operations could occur at this level, retaining as much of the elegance of the mathematics as possible. This was attempted quite successfully previously (Angus and Thompkins 1989).

C++ enables one to design scientific codes in a more modular fashion than is practical to do with FORTRAN, and it allows for much more flexibility in the type of data structures used. A significant and compelling reason to use C++, however, is the natural decomposition of the problem for parallelization. The data and the methods in finite element or finite difference paradigms are kept together, providing all the information for solving the equilibrium equations in the local grids. The development of a class structure that makes maximum use of inheritance for distributed memory is one goal. For example, in a finite element code an element might reside on a single processor of some parallel framework.

This is the second of two papers describing OOP issues for scientific code development. The first paper (Ross et al. 1992) discusses how a scientific problem can be decomposed using object-oriented design for computer modeling. In the present paper three simple scientific codes are used to illustrate the potential of C++. The first is a one-dimensional (1-D) finite element code that addresses the Newtonian equation of dynamic force equilibrium. The code uses a Lagrangian coordinate formulation, and the equation solver is implicit (i.e., the resulting algebraic equations are solved simultaneously). The second is a 1-D finite difference code that addresses computational fluid dynamics (CFD). The code uses an Eulerian coordinate formulation, and the equation solver is explicit (i.e., the difference equation is solved recursively). The third code is a 2-D version of the 1-D CFD code. While these simple codes are for linear 1-D and 2-D physical problems, the object structure provided in the libraries is extensible in a straightforward manner to higher-dimensional and nonlinear problems.

Structural Applications

Finite Element User Code

In order to demonstrate some preliminary object structure formulations, a user code employing the finite element numerical method is presented here for a very simple problem: the displacement of a copper bar attached to a rigid wall on one end and subject to a dynamic force on the other. The following governing equation (ordinary differential equation) represents this structural dynamics problem

$$m \frac{d^2x}{dt^2} + kx = f(t)$$

where m = the mass and k = the stiffness of the rod, x = the spatial coordinate along the rod, t = time, and $f(t)$ = the dynamic force applied.

Fig. 1 is the user code to monitor distortions of a cooper rod due to a dynamic force load. The structural dynamics problem is solved using an implicit-solution method.

In a production environment, the input/output (I/O) for this user code

```
// force.C
//
//     dynamic force on a copper rod
//

#include <stream.h>
#include <finite_element.h>   // Kachina object libraries of Finite Element

main()
{
  ImplicitGoverningEquation   g=NEWTON1; // Newton's eqn of motion in 1-D
                                         // LaGrangian grid is the default

  LagrangianNodeField     x;
  Vector                  f;
  ElementField1           e; // specify a one dimensional problem
  double                  area,time_step;
  int                     i,nodes,steps;

  cin >> area;                 // cross section of the bar
  cin >> time_step >> steps;
  cin >> nodes;
  x.read_locations(nodes);     // reads in location vector for "nodes" nodes
  x[0].boundary();             // specifies node 0 as a boundary where
                               // displacement always equals zero
  e.standard_mapping(x);       // i.e. element i extends from node i to node i+1
  e.cross_section() = area;
  g.increment(time_step);      // override default time step if set in NEWTON1
  e.material(all,M_COPPER_CAST); // all elements are copper
  x.rest(); // initial condition, all nodes are at rest
  f.set_size(nodes); f.assign_constant(0.0);

  for(i=0; i<steps; i++)
    {
     cin >> f[nodes-1]; // apply dynamic load to right end of bar
     g.apply_force(e,f); // apply force to the element field
     x[nodes-1].print_location(); // plot changing length of the bar
    }
}
```

FIG. 1. User Code for 1-D Finite Element Model for Structural Dynamics Problem

should be somewhat different than what was just described. One can not always count on having such nice initial conditions as everything at rest (rest() applied to x) or to have the nodes so nicely arranged (standard_mapping from NodeField x to ElementField e), especially in higher dimensional problems. It is likely that a finite element code like this will have a graphical front end such that the user code is something like cin << bar_window (some parameters). One of the advantages of OOP is the ability to seamlessly add new views to a preexisting object—in this case allowing for graphical descriptions. Such graphical views can be anything from a function that reads in information from a separate preprocessing package and another that outputs to a separate postprocessing package (as is done with most current FORTRAN codes) to routines that do pre- and postprocessing interactively.

Note that nowhere in the user code are k (stiffness) or m (mass) directly mentioned. The material M_COPPER_CAST knows about certain properties of copper, such as density and stiffness. Likewise ElementField e learns the length of each element when the nodes are mapped into it; the (constant) area is given by the user, and e then calculates the volume of each element. The user next gives the material, Copper (Cu), and e can determine mass = density·volume, as well as stiffness = elastic modulus·area/length. The velocity and acceleration at each node point is stored within the LagrangianNodeField x.

Finite Difference User Code

The 1-D heat conduction problem involves a partial differential equation relating the change in temperature, T, to position (x) along the bar as a function of time (t) and the coefficient of thermal diffusivity, k. The equation is of the parabolic form and is given here

$$k \frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t} \quad (1)$$

Fig. 2 illustrates message passing among the objects for this code (arrows indicate information queries). Fig. 3 is the user code for a simple 1-D heat-flux problem along an aluminum rod.

Brief Explanation of Finite Difference Code

Fig. 2 shows how data is hidden in this problem. Each individual node knows its material type and from this the NodeField (EulerianNodeField to be specific) that it is associated with can determine useful extremal properties. In the 1-D example, all nodes are of the same material, but if there were a rod made of two or more materials, the material with the largest coefficient of thermal diffusion would determine the time step for solving the problem.

The thing that is interesting about the inner workings of the finite difference user code shown in Fig. 3 is that the explicit solver knows nothing about boundary conditions. It is a simple recursion equation of the form

$$\text{for all} \quad i \qquad T_i^{t+1} = T_i^t + \lambda(T_{i-1}^t - 2T_i^t + T_{i+1}^t) \quad (2)$$

where λ = the stability number of the problem; the default is 1/2 in HEAT_FLUX_1.

The solver is written to treat all nodes as internal. The *boundaries* vector in the code is composed of one of two values for each node (to designate that the node is either a heat source/sink or internal). The make_temp_boundary() method associates this input vector with the node field. All that is necessary to change the boundary conditions is to alter the data file. The governing equation (and the explicit solver embedded therein) would be left unchanged.

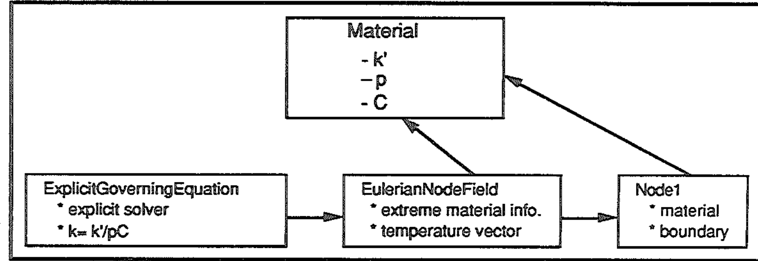

FIG. 2. Message Passing in Finite Difference Code

```
// hback.C
//     heat transfer along an aluminum bar
//         temperatures are in degrees Celsius, length is in cm
//         sensitivity to change is in degrees C per second

#include <stream.h>                   // system I/O library
#include "finite_difference.h"        // Kachina FD libraries

main()
{
  int intervals,num_nodes;
  double sensitivity,deltax,length,lambda,time_checks;
  vector temp,boundaries;
   // k(d2Temp/dx2) = dTemp/dt, k is coef of thermal diffusivity
   // HEAT_FLUX_1 is an explicit forward solver
  ExplicitGoverningEquation g = HEAT_FLUX_1;
  EulerianNodeField   n(TEMP);       // fixed grid, concerned with temperature at
                                     // each node and nothing else

  cin >> length >> num_nodes;
  cin >> sensitivity >> lambda >> time_checks >> intervals;

  n.init_set_size(num_nodes);                      // # of nodes in the field
  n.assoc_field_material(M_ALUMINUM_CAST);  // rod is all aluminum

   // initial temperatures and boundary conditions
  temp.set_size(num_nodes); cin >> temp;
  boundaries.set_size(num_nodes); cin >> boundaries;

  n.init_temp(temp);
  n.make_temp_boundary(boundaries);
  deltax = (double) length / (double)(num_nodes - 1);
  n.set_locations(0.0,deltax);          // location in space of each node

  g.set_lambda(lambda);       // stability #
  g.assoc_node_field(n);      // what equation g will operate on

  cout << form("deltat = %lf, deltax = %lf\n",g.get_deltat(),deltax);
  for(i=1; i<=intervals; i++)
    {
    g.step_for(time_checks);
    cout << form("at %8.2lf seconds:  ",i*time_checks) << n.get_temp();
    }
   // applies explicit solver until no node changes temp by > sensitivity in a
   // single time step (which is dependent on the stability number lambda
  g.converge(sensitivity);
  cout << "Stabilized Temperature at Node Points:\n" << n.print_temp();
}
```

FIG. 3. User Code for 1-D Heat-Conduction Problem Using Finite Differences

The implementation of the recursive equation just shown is by passing messages to the associated node field to change each node according to the formula, and the node field in turn passes a message to the node to change its temperature. If a node is a temperature-boundary node (heat source/sink), it simply ignores the message. While boundary conditions are not difficult to model in 1-D, many 2-D and 3-D problems can be fairly easily discretized at the interior, but dealing with boundaries turns a simple explicit solver into a nightmare in FORTRAN. In this object-oriented system, the discrete solver is simply written as if all nodes were on the interior, and the

boundary nodes modify the operation or cancel the operation themselves as necessary.

Note that the user never refers to k (the coefficient of thermal diffusivity) or Δt (the increment of each time step). The governing equation, g, sets

$$\Delta t = \lambda \frac{\Delta x^2}{k} \quad (3)$$

where λ = stability number and k = largest k in any node.

The governing equation, g, knows the values of Δx and k from querying the node field; the node field determines k by querying the nodes. The stability number, λ, can be changed by the user, but it defaults to $\lambda = 1/2$ in HEAT_FLUX_1. Note that changing the stability number, by using the set_lambda() method (sending the message set_lambda() to the governing equation, g) automatically modifies the underlying time step. The user can check temperature flux over time by using the step_for() method. The steady-state temperature distribution can be found by using the converge() method, which simply applies the solver at each time step. It iterates if the temperature at any node has changed by more than a user prescribed tolerance, otherwise converge() will halt.

Expansion from 1-D to 2-D Finite Difference Codes

Notice that the finite difference user code for a 2-D heat-transfer problem given in Fig. 4 is almost identical to the user code for the 1-D problem (Fig. 3). What is happening in each case is fundamentally the same, but with an added spatial dimension. The logic behind the 2-D code is identical to that of the 1-D code.

In an object-oriented environment, communication is through a message interface that need not reflect the underlying data structure. This allows polymorphism in message passing—using the same name for the same operation on different types of objects and letting the object do the work of interpreting that message correctly. For example, the function print_location() attached to a node "knows" to print only an x coordinate for a 1-D node but to print x and y coordinates if it is attached to a 2-D node.

The solver has no knowledge of temperature-boundary conditions. Boundary conditions are input through the *boundaries* matrix in the 2-D code in the same manner as through the *boundaries* vector in the 1-D code. As in the case with the 1-D code, the solver treats all nodes as internal and sends change_temperature() messages to each node based on this. The node knows if it is a boundary and alters or ignores the message as appropriate.

It is important to note that 2-D nodes (Node2) are derived from 1-D nodes (Node1) and that all information applicable to a 1-D node is inherited by the 2-D node, with certain functions added that apply strictly to the second dimension. Also, the meaning of some functions is overridden in the derived type—such as print_location() just mentioned.

The comments about proper I/O handling given after the finite element code hold for the two finite difference codes. The need to encapsulate I/O details in functions naming the concept becomes more important as the dimensionality of the problem increases.

Object Library Structure

The ultimate goal is to build a C++ library of mathematical and engineering objects that allows for relatively simple user code for scientific

```
// hback2.C
//    heat transfer on an aluminum plate
//       temperatures are in degrees Celsius, length is in cm
//  .    sensitivity to change is in degrees C per second

#include <stream.h>            // system I/O library
#include "fd2.h"       // Kachina FD libraries

main()
{
  int intervals,num_rows,num_columns;
  double sensitivity,deltax,length,deltay,width,lambda,time_checks;
  matrix temp,boundaries;
   // k(d2Temp/ds2) = dTemp/dt, k is coef of thermal diffusivity, s=(x,y)
   // HEAT_FLUX_2 is an explicit forward solver
  ExplicitGoverningEquation2 g = HEAT_FLUX_2;
  EulerianNodeField2 n(TEMP);      // fixed grid, concerned with temperature at
                                   // each node and nothing else

  cin >> length >> width >> num_rows >> num_columns;
  cin >> sensitivity >> lambda >> time_checks >> intervals;

  n.init_set_size(num_rows,num_columns);          // # of nodes in the field
  n.assoc_field_material(M_ALUMINUM_CAST);  // rod is all aluminum

   // initial temperatures and boundary conditions
  temp.set_size(num_rows,num_columns); cin >> temp;
  boundaries.set_size(num_rows,num_columns); cin >> boundaries;

  n.init_temp(temp);
  n.make_temp_boundary(boundaries);
  deltax = (double) length / (double)(num_rows - 1);
  deltay = (double) width / (double)(num_columns - 1);
  n.set_locations(0.0,deltax,deltay);            // location in space of each node

  g.set_lambda(lambda);       // stability #
  g.assoc_node_field(n);      // what equation g will operate on

  cout << form("deltat = %lf, deltax = %lf,deltay = %lf\n",
                    g.get_deltat(),deltax,deltay);
  for(i=1; i<=intervals; i++)
    {
    g.step_for(time_checks);
    cout << form("at %8.2lf seconds:  ",i*time_checks) << n.get_temp();
    }
   // applies explicit solver until no node changes temp by > sensitivity in a
   // single time step (which is dependent on the stability number lambda
  g.converge(sensitivity);
  cout << "Stabilized Temperature at Node Points:\n" << n.print_temp();
}
```

FIG. 4. User Code for 2-D Heat-Conduction Problem Using Finite Differences

applications. This would allow a good scientist who was also a naive programmer to program directly from representations he or she is comfortable with and count on the underlying libraries to make this efficient.

Fig. 5 shows the conceptual framework for some of our libraries. The arrows indicate class derivation. The creation of general libraries reduces the work necessary to expand the user code from 1-D to 2-D and 3-D problems.

A properly designed object library has a robustness that can not be matched by FORTRAN. The changes to the user code necessary to calculate heat flux in 2-D are accomplished by simply specifying 2-D nodes in the node field, g = HEAT_FLUX_2 in the governing equation, and telling the node field how many nodes it has in each dimension.

Fig. 5 is included to provide a feel for how somewhat different problems would be handled. The lowest-level object conceptually is the material model.

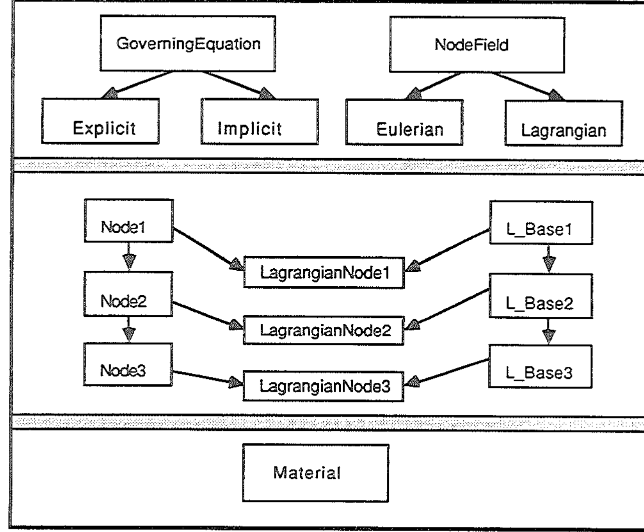

FIG. 5. Objects and Object Hierarchy in the Finite Difference C++ Library

It allows you to describe a material, give it a name, and define certain properties of the material (e.g., density, coefficient of expansion, etc.). Properties of a material are thus encapsulated by its name (e.g., M_ALUMINUM_CAST in the finite difference code examples).

Nodes inherit information from parent classes (e.g., Node2 inherits from Node1) as well as from Material (i.e., a node knows its material composition). The basic node is a 1-D Eulerian node (Node1). A 2-D Eulerian node (Node2) is just a 1-D node (Node1) with a few extra variables and methods, and a 3-D node is likewise derived. Lagrangian nodes are similar to Eulerian nodes with displacement (and rate of displacement) data and methods added. Some methods have slightly different meaning based on dimensionality (e.g., set_locations() has differing components to a location) and these are simply rewritten at each level. Since they maintain the same name, the user code does not change.

The highest-level objects in these libraries are governing equations and node fields. Node fields utilize properties encapsulated in the nodes associated with these fields. The current library has a rather ugly implementation of node fields mainly due to type restrictions within class declarations (solved by the introduction of templates, described later in this paper).

Since governing equations are conceptually operators on objects like node fields, it might at first seem more intuitive to model them as messages rather than as objects. However, this would require designing multiparameter messages with variable relationships among these parameters in order to deal with equations in general. The writers view governing equations as objects in their own right that pass messages to and from the node fields.

Optimizing OOP Code

OOP versus FORTRAN vis-a-vis Optimization

A common complaint about OOP techniques is that they are most valuable in an idealized computing environment. The argument is that all the

gains in clarity and code reuse come at a cost in resources, such as memory and CPU cycles, that is prohibitive for serious scientific codes. Indeed, there are important differences between programming for a business community with inexpensive computing power far in excess of its basic needs and programming for cutting-edge scientific codes that strain the limits of the most advanced hardware. The OOP community has been very slow to address these differences.

It is necessary to distinguish between the writers of user code and the writers of library code. This distinction is made in the FORTRAN community (not everyone writes LINPACK routines), but it has added power when applied to an OOP language such as C++. As mentioned previously, the scientist writing user code in an OOP language need not know or care about the underlying data structure, and, as shall be seen, the user code programmer need not even be fully cognizant of what kind of routines are being called or even when such routines are called.

The life of a library-code designer/modifier is not so carefree. The OOP community has not fully come to grips with this fact, often relying on future compiler technology to solve any problems. Yes, there is increased code reuse even at this level, and, yes, modularity and encapsulation make an OOP language library code easier to modify than a FORTRAN library code, but in most cases greater care is needed in initial design of an OOP language style library than a FORTRAN style library. Some problems will be most naturally solved at the compiler level, and the solution will require compilers that are in some ways "smarter" than FORTRAN needs to be, but it is necessary to isolate what compiler techniques can be reasonably implemented now in order to justify the use of OOP in cutting edge applications now.

There are two key problems in optimizing OOP code:

1. While OOP lets the programmer treat conceptual objects as fundamental data types, it does not necessarily let the compiler recognize, and make use of, the properties of these objects in the same manner as a hard-wired data type like an integer.
2. In FORTRAN, a system being modeled must be treated as an "object of arrays" whereas an OOP language allows the more natural view of an "array of objects."

Unfortunately, it is often easier for a compiler to do optimization at the array level, and in some cases it may be wise to develop tools that allow a compiler to "see" that an "array of objects" can also be viewed as an "object of arrays" and do optimization accordingly. While this is in some sense a subset of the first problem, it is important enough, and solvable enough, to deserve special attention.

Fundamental Data-Type Example

In C++, two matrices can be added together and stored in a third by the simple and elegant construction:

$$\mathbf{A} = \mathbf{B} + \mathbf{C} \qquad (4)$$

which brings to mind the way a programmer would add two integers and store the result in the third:

$$a = b + c \qquad (5)$$

but, in terms of a compiler's ability to optimize, these statements are not at all similar! In (5), the compiler knows what the operators + and = mean when applied to integers and can compile this statement into

```
move  c,a   move contents of c into a
 add  b,a   add contents of b to contents of a and put result in a
```

but, in instruction (4), the compiler does not know the semantics of + and = and must evaluate this statement from the bottom up. (Recall that + and = on matrices invoke methods defined outside the scope of the compiler.) A naive implementation of matrices and the operators + and = in C++ will compile into something of the form:

```
 add  B,C,TEMP   put contents of B+C into TEMP
move  TEMP,A     move contents of TEMP into A
```

Since **A**, **B**, and **C** could easily be huge matrices, this extra copy could be very expensive. Note that adding three matrices (**A** = **B** + **C** + **D**) would result in the creation of two temporaries along the way.

There are several methods of solving this particular problem. One might be to have operators such as + and * do nothing but build expression trees that operator = solves. As an example, for **A** = **B** + **C** + **D**, operator = would be handled as shown in Fig. 6, and it could be evaluated from the top down in the same manner that the compiler would solve (4). This is probably not the most elegant way of dealing with this problem, but it is presented here because it motivates an example of inherent OOP efficiency later in this article.

This case could be dealt with by semantic constructions in the language (Cline and Lea 1990) that allowed the programmer to specify fundamental properties of an object (e.g., matrix addition is associative and commutative, A = (expr) assigns to A), but it is not at all clear that compilers that can handle such things well in the general case will be around any time soon.

Object of Arrays versus Array of Objects Example

Suppose some physical system is being modeled by a finite difference method. Conceptually, there exists a node field where each node has several

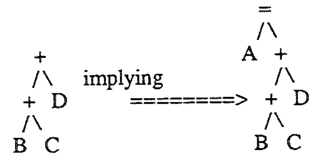

FIG. 6. Expression tree for A = B + C + D

FIG. 7. Data Layout for Example in FORTRAN Arrays and OOP Array of Objects

attributes such as temperature, force(s) acting on it, material composition, and properties, etc. FORTRAN forces this to be programmed as several arrays (e.g., TEMP, FORCE1, FORCE2) with operations being coded in terms of these arrays. An OOP language would allow us to create a node field made of nodes where each node has various properties—closer to intuition. Now, consider a piece of code to modify the temperature of the nodes in this system. In FORTRAN, this would act on the array TEMP, whereas in an OOP language it would apply temperature changes to the node field. Further consider the data layout implicit in this example, as illustrated in Fig. 7.

Suppose this piece of code is running on a serial or vector machine with a large data cache. In the FORTRAN case, it is fairly straightforward for the compiler to throw the TEMP array into the cache for the span of the code that modifies temperature. It is not at all trivial to do the same thing in the OOP language "array of objects" case. How does the compiler know to pluck out a particular member of each object, line these members up in the cache, and then proceed to execute the code?

This could be done with some modification at the node field level—actually laying out storage so that members stay together and altering accessor functions so that they resolve to "object of array" form. This loses a lot of the benefits of OOP. When taken to the extreme, it is little more than writing FORTRAN in a more trendy language. The user code remains clean, but the library-code writer is now moving towards a FORTRAN coding style. This is an example of an efficiency problem that should at least be partially solved at the compiler level.

Actually, because of modularization in OOP languages like C++, it is not too difficult to start with an inefficient yet clean object structure, and add efficiency improvements without breaking down the structure. Our libraries originally specified nodes as just described, with node fields being instantiated as a vector of nodes with some controlling parameters (e.g., size) and methods (e.g., print_locations). Later, temperature and force components were removed from the nodes and equivalent vectors were added to the node fields. This was done originally to save space, but it solves this problem as well. (The declaration EulerianNodeField *n* (TEMP) means to declare a node field that will worry about temperature, but not force. When the size of the node field is established, space is reserved for a vector of temperatures but not for a vector of forces.)

As discussed earlier, our original conception of the internals of changing temperature was as follows: tell the node field to change temperature according to some rule; the node field tells each node to change its temperature treating all nodes as internal, the node then checks to see if it is internal or a boundary—applying, modifying, or ignoring the message depending on its boundary status. When temperature is moved out of the nodes, the node field's change_temperature routine must now be modified to query each node as to its boundary state and proceed accordingly.

This moves away from the intuition of the OOP method. While it is still cleaner than FORTRAN, and while this change results in no modification to user code, it makes life a little tougher for the implementer of library code. In particular, temperature data layout and routines will require modification on two types of objects (nodes and node fields) in order to be efficient on an MPP architecture.

Advantages of OOP Languages in Optimization

In optimization, all is not merely a game of catchup with FORTRAN. For example, consider the expression evaluator mentioned for matrices just presented. Such an expression evaluator has the potential to not only catch up with but actually surpass the efficiency of the FORTRAN matrix operation by function call paradigm. For example, if we have $\mathbf{A}^{s\times b} = \mathbf{B}^{s\times b} * \mathbf{C}^{b\times s} * \mathbf{D}^{s\times b}$ where s is small and b is big, an expression evaluator could know to change $\mathbf{B}*(\mathbf{C}*\mathbf{D})$ to $(\mathbf{B}*\mathbf{C})*\mathbf{D}$ because it provides the same result with fewer computations. In FORTRAN, if there exists a subroutine

MATMUL (Product, Left-Matrix, Right-Matrix, Rows-in-Left-Matrix, Columns-in-Left-Matrix, Columns-in-Right-Matrix)

to multiply two matrices, the user needs to know that

```
MATMUL(Tmp1,B,C,s,b,s)
MATMUL(A,Tmp1,D,s,s,b)
```

is superior to

```
MATMUL(Tmp2,C,D,b,s,b)
MATMUL(A,B,Tmp2,s,b,b)
```

By the same token, a matrix could "know" many properties about itself such as whether it is sparse. A matrix input routine could cheaply test for this property and the library could determine which matrix multiply routine to call depending on whether the operands are sparse. The user need not know about the efficiency advantages of sparsity—the code would just run faster.

Also, lets consider a machine with a large data cache as in the "object of arrays" example, but this time code is being constructed that acts on a number of attributes of a node all at once, and spends a lot of time at each given node. Since C++ (for example) has control of its own memory management, a programmer could rewrite the object creation/delete routines to align objects properly in the cache. This is an extremely straightforward operation in C++, but extremely difficult (at the library OR the compiler level) in FORTRAN.

C++ as OOP Language for Scientific Codes

C++ was the language of choice for our research (Wagner et al. 1991) in part because it is *not* a pure object-oriented language. One of the key design goals of the language can be stated as "you should not have to pay (in storage, CPU time, etc.) for what you do not use." Most OOP languages were designed with the overriding goal of mirroring a clean object-oriented paradigm, without regard for efficiency at the language design level, and programmers pay an efficiency penalty for this.

One nice feature of C++ is the ability to inline small functions. Consider the problem of adding two matrices that are stored as a series of row vectors. A straightforward implementation of the + function would be of the form:

```
matrix operator + {matrix& A, matrix& B}
{
  for { 0<=i<=rows}
    Sum[i] = A[i]+B[i];
  return Sum;
}
```

where $A[i]$ invokes the [] operator of the matrix class, which returns the ith row vector associated with A, and the line $A[i]+B[i]$ invokes operator + of the vector class. The operator + for vectors would look similar, making calls on operator [] for the class vector. Conceptually, if a user types $A+B$, this invokes the function + for matrices, which invokes the matrix [] function and the vector + function for each row in A and B. The vector + function will then invoke vector [] function for each column element. Thus, a simple statement $(A+B)$ can represent a lot of potentially expensive function calls for very little actual code. Adding the keywork *inline* in front of a function definition requests that the compiler inline expand the code rather than make a function call. In the example above, inlining the appropriate functions means that the user code $A+B$ will compile to a simple nested loop with no function calls. Whether or not a message is a function call or inline expanded has no effect on the semantics of calling that message. This eliminates function call overhead at run time, but can lead to increased object file size if applied to large functions.

For a more fundamental example, default name resolution in C++ is static, which is to say that objects and messages between them are bound at compile time. In a pure OOP language (e.g., Smalltalk) name resolution is dynamic, which is to say binding is done at run time and thus incurs a run time penalty. An overall goal of C++ is to "do the object-oriented material that can be done at compile time," therefore bypassing many of the run-time costs involved in the object-oriented paradigm. So long as name resolution is static, good software tools can determine many useful things at compile time. For example, they could determine that many functions within an object semilattice will not be accessed by a particular piece of code, thereby greatly reducing the size of object files. In many object-oriented languages, name resolution is far more (perhaps totally) dynamic, and such optimizations are far more difficult to implement.

Dynamic name resolution, is, however, a very powerful feature of OOP, and C++ does allow this, after a fashion. By declaring a class or function to be "virtual," the user is directing the program to determine the relevant typing information at run time. If the user derives everything from a virtual base class, he has gained much of the power of the totally dynamic system, but he has lost much of the optimization ability.

Since C++ lets the user choose the type of name resolution, and the compiler knows what is and is not statically resolved, this largely allows the builder of the library to determine whether these efficiency trade-offs are desirable. When incorporating other people's work, however, different views of the importance of efficiency may lead to performance trade-offs that are not apparent to the user. (In other words, class browsers should access comments about efficiency and other considerations involved in the creation of various object libraries.) This problem is by no means unique to object-oriented methods.

Run-time Debugging

One of the pleasures of writing in C++ is that, because of the inherent modularization, it is obvious where to test for errors in function usage and it is natural to do so. An example of the usefulness of this is given here. If one improperly mixes matrices in a FORTRAN package there is likely to be some cryptic message from the assembler, if not simply incorrect results, whereas in C++ one will get something of the form

"In matrix multiply (operator*) column size of first matrix is unequal to row size of second."

While this is very nice for debugging purposes, it does present two immediate problems.

- This C++ message does not provide the user with a trace of exactly where and in what state the error occurred. If the user is relying on a C++ to C translation, the debugger might not give this information cleanly.
- This run-time checking slows the user down. If a matrix is made up of vectors, similar checks may well occur at the matrix and vector class level for all matrix operations. In some instances, this checking could quite be time-consuming.

These problems could be solved by a modification to C++ that allows "semifree" run-time debugging. When writing matrix classes, a class of commands that could have access to stack information (to provide a state trace) and that were switchable at compile time so that no run-time penalty would be incurred in production code is desirable. The method used now is to wrap run-time checking with C preprocessor directives, a method that is neither pretty nor robust.

The language Eiffel has addressed these two problems in an elegant and efficient manner (Meyer 1989). A routine is thought of as a contract between the caller and the callee. It has associated with it preconditions, which must be satisfied by the caller, and postconditions, which must be satisfied by the callee upon return to the caller. A vector-add operation might have precondition (sizes are equal) and a postcondition (true). Condition checking is switchable at compile time. There is no problem with cross-compiling libraries where some have condition checking and some do not.

An Eiffellike solution could be easily implemented in the same form in C++, but there may be trace problems when using C++ to C translators. At least one version of C++ incorporating Eiffellike constructs has been implemented (Cline and Lea 1990). Given that many small functions are inlined by C++ (to avoid the cost of a function call) tracing where something "really" occurred in the C++ source could be difficult. A slightly modified form could be to provide the equivalent of C's_LINE_and_FILE_ directives and let the programmer write a precondition with failure directives (generally, print error message and halt) having access to the "real" location of the call in the C++ source. The problem is essentially one of exception handling, and language constructs for C++ to deal with exception handling are being debated by the ANSI C++ Standards Committee now.

Templates

A template (also known as a parameterized type) allows one to apply a type or types to a class declaration. The type specific class, as well as all type specific member functions, are automatically generated at compile time from the templates. For example

```
vector<T> {...
<T>   *data
...
}
```

could be the source code (template) for a generic vector object of unspecified type, while

```
vector<int>   ivec;
vector<Node2>   nvec;
```

would direct the compiler to generate the class and associated functions for a vector of integer and a vector of 2-D nodes automatically. C++ afficionados will note that the declarations in our user codes (Figs. 1, 3, and 4) are not in template form. This is because we did not have a version of C++ with true templates when these codes were first written.

Ideally, each conceptual object would be a C++ class, with each derived class defining only those functions and data specific to it, while inheriting the more general information from ancestor classes. The node C++ classes map in exactly this manner. A 2-D Eulerian node (node2) inherits all the properties of a 1-D Eulerian node (node1) plus various data and methods pertaining to the second dimension. In addition, node2 overwrites those functions whose meanings have changed slightly (e.g., read_location).

However, the *node field* objects are handled in a far more cumbersome manner. The lack of templates in the C++ implementation used over the course of this project led to some "ugly code" for 1-D node fields as well as a rather bizarre extension to 2-D node fields. Our libraries contain several routines with identical code within EulerianNodeField1 and LagrangianNodeField1, which are both derived from NodeField1 (the 1-D node field structures). Conceptually, the user would want to write these routines once in NodeField1 with the two derived types inheriting these routines. The reason this can not be done under the current implementation is that the routines reference a vector of nodes attached to the fields, and the type of node depends on the type of field. This problem is compounded by additional layers of object dependencies within a type hierarchy, such as the dependence of governing equations on the type of the associated node field. Also, the 2-D node fields should be derived from 1-D node fields in the same way that 2-D nodes are derived from 1-D nodes. Because of the code duplication problems presented by these embedded types, the *node field* C++ classes are entirely new structures for each dimension. (i.e., 2-D node fields are specified in their entirety rather than inheriting those characteristics common to 1-D node fields)

It would be helpful to declare something of the form,

```
NodeField<Node2>   or   NodeField<LagrangianNode1>
```

and have it generate the body of

```
EulerianNodeField2   or   LagrangianNodeField1
```

which allows node fields to inherit data and functions that have the same basic outline with the only difference being the type of node they operate on; in other words, a certain amount of polymorphism would be beneficial. This becomes even more important when a class is doubly type dependent, such as with the governing equation hierarchy. Experience in writing these object libraries has led us to the conclusion that templates are essential to realizing code reuse and maintenance goals. Since our user codes were first written, templates have been fully incorporated into C++.

Summary

Several current OOP languages were designed under the assumption that computing power is cheap while programmer time is expensive and growing

more so as computers are allotted more complex tasks. Unfortunately, computing power is very expensive in the domain of large scale scientific codes and likely to remain so for the near future. It is our belief that C++ can be made comparable in efficiency to FORTRAN using established compiler writing techniques (Wagner et al. 1991).

Since run-time efficiency is desired without waiting for the development of compilers to solve all optimization problems, the job of the library code programmer, as opposed to the crafter of user code, will remain more difficult than some OOP proponents believe. Still, OOP is an improvement over previous techniques even at this level, and it should be noted that a user code/library code dichotomy that makes creation, modification, and efficient porting of user code much easier is possible and even natural using OOP techniques. It is the authors' opinion that a great deal of the effort involved in crafting user code for large scientific applications is a direct result of the weak distinction between user and library code given current programming practices and languages used in this domain.

Although much remains to be done to make OOP a more realistic and useful paradigm for scientists and engineers in their attempts to model the physical world, the few works cited in this field have shown tremendous potential. Although there has been a great deal of work in other OOP languages (e.g., Smalltalk, Eiffel, CLOS), recent works in scientific code development using OOP environments have concentrated primarily on C++. An earlier work (Angus and Thompkins 1989) reports on the efficiency of "naive" C++ code recompiled for various serial and parallel architectures versus C code optimized for each environment and shows the C code to be more efficient by factors ranging from 1.5 to 3. In addition to this, another work (Forslund et al. 1990) cites that the efficiency of FORTRAN over C++ varies between a factor of 1.7 for nonoptimized FORTRAN to a factor of 2.2 for vectorized FORTRAN for a single-processor architecture. Although these factors may seem disappointing for OOP proponents, they actually bode tremendous potential for C++ given the facts that:

1. The C++ user code can be considerably less than FORTRAN user code and this ratio becomes more pronounced as codes become more complex.
2. The C++ user code is highly modular with a library of reusable objects.
3. The granularity of C++ code objects can be controlled to be consistent with the new MPP architectures without changes in user code.
4. Dynamic load balancing among processors for MPP architectures will prove to be easier for OOP codes because the structure allows for the dynamic reconfiguration of numerical grids.
5. Current C++ compilers are in their infancy and improving rapidly (Wagner et al. 1991).

Acknowledgment

The writers are grateful to the Air Force's Phillips Laboratory for sponsoring this research under contract F29601-90-C-0046. The writers wish to thank Dr. Ian Angus, Boeing Computer Services, for his early discussions and results; Drs. David Forslund and Stephen Pope of the Los Alamos National Laboratory; and Mr. Paul Morrow of the Phillips Laboratory for their comments and suggestions during the course of this research. We are

also grateful to Dr. Robert Ballance, Kachina Technologies, Inc., for his thoughts on optimization of OOP codes.

APPENDIX I. REFERENCES

Angus, I. G., and Thompkins, W. T. (1989). "Data storage, concurrency, and portability: an object oriented approach to fluid mechanics," *Proc. 4th Conf. on Hypercubes, Concurrent Computing, and Applications*, Institute of Electronic and Electrical Engineers, Mar.

Cline, M. P., and Lea, D. (1990). "The behavior of C++ classes," *Proc. Symp. on Object Oriented Programming—Practical Applications*, Mavist College, Sep. 81–91.

Filho, J. S. R. A., and Devloo, P. R. B.(1991). "Object oriented programming in scientific computations: the beginning of a new era." *Eng. Comput.*, 8(1), 81–87.

Forslund, D., Wingate, C., Ford, P., Junkins, S., Jackson, J., and Pope, S. (1990). "Experiences in writing a distributed particle simulation code in C++." *Proc. 1990 Usenix C++ Conf.*, Usenix Association, 117–190.

Lu, Z. L., and Ross, T. J. (1991). "Diffusing-vortex numerical scheme for solving incompressible Navier-Stokes equations." *J. Computational Physics*, 95(2), 400–436.

Meyer, B. (1989). "Writing correct software in Eiffel." *Dr. Dobb's J.*, 14(158), 28–36.

Peskin, R. L., and Russo, M. F. (1988). "An object-oriented system environment for partial differential equation solution." *Proc. ASME Computations in Engrg.*, American Society of Mechanical Engineers, 409–415.

Ross, T., Wagner, L., and Luger, G. (1992). "Object oriented programming for scientific codes: thoughts and concepts." *J. Comp. in Civ. Engrg.*, ASCE, 6(4), 480–496.

Wagner, L., Luger, G., and Ross, T. (1991). "Object-oriented programming in C++ on the Cray for scientific codes." *Tech. Report PL-TR-91-1037*, Phillips Laboratory, Kirtland Air Force Base, Albuquerque, N.M.