FastScat™: An Object-Oriented Program for Fast Scattering Computation

LISA HAMILTON, MARK STALZER, R. STEVEN TURLEY, JOHN VISHER, AND STEPHEN WANDZURA

Hughes Research Laboratories, 3011 Malibu Canyon Road, Malibu, CA 90265

ABSTRACT

FastScat is a state-of-the-art program for computing electromagnetic scattering and radiation. Its purpose is to support the study of recent algorithmic advancements, such as the fast multipole method, that promise speed-ups of several orders of magnitude over conventional algorithms. The complexity of these algorithms and their associated data structures led us to adopt an object-oriented methodology for FastScat. We discuss the program's design and several lessons learned from its C++ implementation including the appropriate level for object-orientedness in numeric software, maintainability benefits, interfacing to Fortran libraries such as LAPACK, and performance issues. © 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

Current problems of interest in computational electromagnetics include the prediction of radar cross sections and the modeling of antenna radiation patterns (see Fig. 1). Methods for computing electromagnetic scattering and radiation generally involve the solution of a matrix equation derived from the discretization of an appropriate integral equation [1]. The matrix equation is often written $Z \cdot I = V$, where the impedance matrix Z depends on the geometry and composition of the scattering or radiating surface, I is a vector containing the expansion coefficients of the current density over the surface, and the excitation vector V represents a dual expansion of the current. The number of unknowns, N, required for accurate modeling of such problems is very large, and, in the past, has severely limited problem size and solution accuracy.

There are two primary areas of difficulty in conventional solutions of these problems. The first is accurate computation of the Z matrix elements. In general, each element of the $N \times N$ matrix requires numeric integration of a function that is often singular on portions of the surface. The second difficulty is the actual solution of such a large matrix equation. This has been done by direct decomposition of the sometimes ill-conditioned Z matrix ($\mathbb{O}(N^3)$ time), or alternatively by iterative methods requiring repeated matrix-vector multiplications ($\mathbb{O}(N^2)$ time for each step).

Recently, a technique called the fast multipole

Received April 1993
Revised June 1993

FastScat™ is a trademark of Hughes Aircraft Company.

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-91-C-0064. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereron.

© 1994 by John Wiley & Sons, Inc.
Scientific Programming, Vol. 2, pp. 171–178 (1993)
CCC 1058-9244/94/040171-08

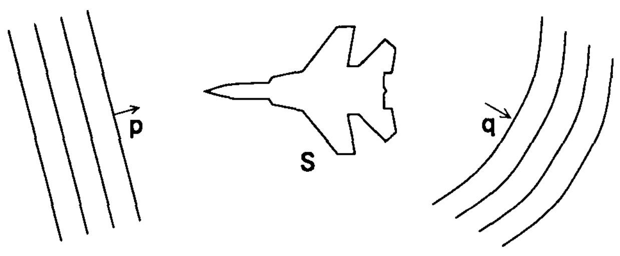

FIGURE 1 Model scattering problem. An incident plane wave $\vec{p}$ (excitation) induces a current on S which re-radiates as the scattered wave $\vec{q}$.

method (FMM) was discovered, which essentially factors the Z matrix into sparse components [2–5]. With this representation, the matrix-vector multiplications required by iterative solvers can be done in $\mathcal{O}(N \log N)$ time. Thus, total solution time is greatly reduced, allowing the study of much larger objects.

Our ongoing effort is to develop a code capable of accurately computing scattering and radiation from surfaces of arbitrary shape and size, represented in either two or three dimensions. In this program, called FastScat, we are implementing conventional solution techniques as well as new computational algorithms, such as the FMM. We also plan to incorporate the ability to scatter from dielectrics and other materials, and to efficiently treat periodic bodies. In addition, FastScat is being used as a testbed to determine the effectiveness of various enhancements such as more accurate surface models, higher order expansion (basis) functions, and more accurate quadrature rules.

To support this work, FastScat must be written in such a way as to be highly modifiable and extensible, as well as reasonably efficient. Specifically, we require a design methodology and language support that can provide a clear implementation of the algorithms and a sensible structure for the underlying data. Our experience in modifying an existing program written in Fortran demonstrated that this, mostly procedural, code lacked important elements needed to incorporate the features described above. Instead, we have turned to an object-oriented methodology [6] in which features such as inheritance, data encapsulation, polymorphism, and dynamic binding allow the key elements of the problem to be expressed and manipulated in a more natural way.

2 AN OBJECT-ORIENTED DESIGN

The design of FastScat is based on the key abstractions of the physics of scattering. In the object-oriented paradigm, a class is used to define a new data type and encapsulates not only the operations that can be performed on that type (methods), but also the implementation or actual data structure of the type. Defining classes to model the physics of the problem provides a clear mapping of the theory and algorithms onto the resulting computer code. For example, the FastScat classes `Surface`, `Z_Matrix`, `Current`, and `Excitation` come directly from the problem formulation given in Section 1. Once defined and implemented, the manipulation of these new types is straightforward and can closely resemble the original equations from physics, thus improving code readability. Using this approach, we have found that when a new class or method seemed awkward or difficult to add it often did not adequately model the physics. As an added benefit of object-oriented thinking, we have sometimes gained a better understanding of physical or theoretical relationships in the problem. On occasion, difficulties in implementation have directed us to a flaw or gap in our physical understanding rather than with the design. The remainder of this section describes some of FastScat's design and the resulting maintainability benefits.

2.1 Modeling Surfaces

In FastScat, the scattering surface or antenna (scatterer) is described using a collection of elementary surfaces. In the current version of FastScat, the elementary surfaces are limited to patches. In two dimensions, a patch is simply a curve in a plane, and in three dimensions, it is a surface. The simplest 3d patch is a flat triangle.

The `Surface` class hierarchy (Fig. 2) provides support for FastScat's surface description. Class `Surface` is abstract and defines basic operations required for all surfaces. These basic operations, which include translate, rotate, scale, and reading/writing, must then be implemented in descendent classes of `Surface`.

A collection of surfaces is maintained by `Composite_Surface`, which descends from `Surface`. An element of a `Composite_Surface` is itself a `Surface`. This organization makes it easy to implement many methods, and permits modeling of hierarchical scatterers. For example, the

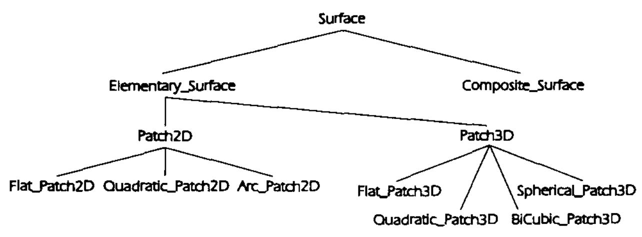

FIGURE 2 Surface class hierarchy.

translate method in `Composite_Surface` simply calls the translate method of each of its elements. An ancillary class supports iteration over all of the elements in a composite surface.

Ultimately, the surface is described in terms of instances of class `Elementary_Surface`. This class, which is derived from `Surface`, currently has two descendants, `Patch2D` and `Patch3D`. In the future, the descendants `Wire2D` and `Wire3D` will be added to support the modeling of wires. The patch classes define several methods. For example, in `Patch2D`, there is a method called `map`, which takes a single parameter $u \in [0, 1]$ and returns a `Vector` to the corresponding point on the patch. The endpoints of the patch are at $u = 0$ and $u = 1$. Another method is `tangent`, which returns the tangent to the patch for a given u. A parallel set of methods is defined by `Patch3D`, except the parameters are u and v. These methods are used by many calculations in FastScat. The important point is that most of the FastScat code is written in terms of `Surface`, `Composite_Surface`, and `Elementary_Surface` objects. The underlying surface model, 2d or 3d, flat, curved, etc., is hidden from most of the code. This eases maintenance and the addition of new features.

2.2 Modeling the Physics

The basic principle behind FastScat's design is to model the physics as closely as possible. The common object-oriented approach is to identify the entities in the problem and proposed solution and to model these using classes. The `Surface` class hierarchy was designed using this approach. Modeling some of the physical concepts is more abstract. Some entities, such as a plane wave, are simple. For a plane wave, we defined a class that contains the wave vector $\vec{k}$ and provides a method to evaluate the wave at any point in space.

A key physics abstraction is a `Surface_Function`. It is defined on the surface of the scatterer and maps a particular location on the scatterer to a tensor. The `Current` and `Excitation` classes are descendants of `Surface_Function`. Various operations are supported on surface functions, including addition, scalar multiplication, and inner product. These operators are used extensively in FastScat's calculations. Although the `Surface_Function` class is currently implemented using class `Array` described in Section 3.2, this representation can and will be changed in the future to implement a different method (Nyström) of discretizing the integral equation.

Closely related to `Surface_Function` is `Surface_Operator`, which maps one surface function onto another. The mapping is performed by the `apply` method. An important example of a `Surface_Operator` is `Z_Matrix` (Z) which takes a `Current` (I) and maps it into an `Excitation` (V). Another example is the FMM, which is implemented in the `FMM` class.

The system $V = Z \cdot I$ can be solved directly using LU decomposition if Z is dense, or by using an iterative solver. Iterative solvers can be used for both dense (`Z_Matrix`) and sparse (`FMM`) surface operators. The iterative solvers are written in terms of `Surface_Operators` and `Surface_Functions`. When support for the FMM was added to FastScat, we only had to concentrate on the details of the FMM as encapsulated by class `FMM`. The solvers did not require modification because they are defined at a higher level of abstraction. The maintainability/extensibility benefits of FastScat's design are discussed further in the next section.

FastScat also contains a class hierarchy for modeling basis functions, which is conceptually similar to that of the surface classes. There is a top-level abstract class `Basis_Function` with descendants for two and three dimensions (`Basis_Function2D` and `Basis_Function3D`). Descendants of these two classes describe particular basis functions, such as Legendre polynomials.

2.3 Maintainability/Extensibility Benefits

One of the major objectives of FastScat was the implementation of the FMM. In the previous section we mentioned how the FMM fit easily into the program's design. This design is also helping to achieve many of FastScat's other objectives. For example, to support different surface models, it is only necessary to add a new descendant to

Patch2D or Patch3D. This flexibility has allowed us to study the importance of higher order surface models for accurate scattering calculations, and has also turned out to be very useful for verification. For a few special geometries, like circles and spheres, the cross section can be computed analytically. In FastScat, a circle can be approximated using flat patches. As the the number of patches increases, so does the solution accuracy. However, even using as many as 1,000 patches only results in a few digits of accuracy in the cross section. Our response was to add a descendant of Patch2D, called Arc_Patch2D, which represents a wedge of a circle. We used the arc patches to construct a perfect model of a circle and were able to compute answers accurate to 11 significant digits.

The structure of the basis function hierarchy allows for similar flexibility. FastScat was originally implemented in terms of pulse (constant) basis functions. Moving up to higher order basis functions was trivial; we simply generalized the pulse basis functions to Legendre polynomials. The rest of the program was unchanged.*

There have been times when it was difficult to use a FastScat component. We have found this with the iterative solvers—they depend on Surface_Function and Surface_Operator, which in turn depend on Surface. Use of the solvers then requires a substantial amount of FastScat code, indicating a flaw in the design. The solvers should have been defined on classes more general than Surface_Function and Surface_Operator, namely Function and Operator. The surface versions would then just be subclasses of the more general versions, and the solvers could be used independently of FastScat by defining the appropriate functions and operators.

3 LESSONS FROM A C++ IMPLEMENTATION

The design of a program is independent of its implementation. In principle, one can have an object-based design and implement it in a traditional language (as is often done with Ada [7]). However, to get full benefit of the methodology, we chose to use an object-oriented language as well.

Pure object-oriented languages, like Smalltalk [8] and CLOS [9], have a high overhead due to their generality, and are not commonly available on supercomputers. C++ [10] has the basic features necessary (such as classes, inheritance, and dynamic binding) for an object-oriented implementation. Because it has been implemented as a translator into C, the language is portable and is widely available on supercomputers. This combination of features and availability led us to the choice of C++.

This section presents some of the lessons we learned from implementing FastScat in C++. Most of what follows is related to performance issues: how to arrange C++ programs so that they run efficiently. We also discuss some of the limitations of C++.

3.1 Overhead of Object-Orientedness

The object-oriented facilities in C++ require runtime support not needed in languages like Fortran. If not properly addressed, this overhead can seriously degrade performance. With our present C++ compiler, the dynamic binding associated with virtual functions takes twice as much time as a regular function call. Consider, for example, the descendants of class Patch2D described previously. Each patch must define the method map, which takes a parameter u and returns a Vector on the surface of the patch. For flat patches, this is a very simple computation and executes in less time than the virtual method call and return. Using inlined methods (type-checked macros) is no help because virtual calls cannot be expanded. However, as illustrated below, there is a simple solution that has the performance of an inline method, the generality of virtual methods, and gives the compiler an opportunity to perform aggressive optimizations.

We often use variations of Gaussian quadrature to perform our integrations. The basic form of a Gaussian quadrature to approximate the integral I of a function $f(x)$ over some region is

$$I \approx \sum_{i=0}^{N-1} w_i f(x_i),$$

where the w_i are weights and the x_i are sampling points (abscissae) for f. Assume we want to integrate the magnitude of the map vector over a

* It is not quite as simple as this. We had to plan ahead and put a method in the basis function class that returns the order of the quadrature required to exactly integrate the function. If we had not, we would have lost accuracy by moving to higher order basis functions.

patch. An obvious C++ implementation is

```
double A1(Patch2D& p) {
  double sum = 0;

  for (int i = 0; i < N; i++)
    sum += w[i]*mag(p.map(x[i]));

  return sum;
}
```

Although simple, this code runs slowly compared with equivalent inlined code due to the overhead in the virtual function call `p.map`. A solution to this problem is to add a `map_all` method that takes a list of places at which to evaluate `map`. The actual implementation is as follows:

```
Class Flat_Patch2D : public Patch2D {
public:
  Vector2D map(double u)
    { return v1 + u*delta; }
  void map_all(int N, double* u,
  Vector2D* results);
  ...
private:
  Vector2D v1, delta;
  ...
};

void Flat_Patch2D::map_all(int N,
double* u, Vector2D* results){
  for (int i = 0; i < N; i++)
    results[i] = map(u[i]);
}
```

The equivalent of function A1 is then

```
double A2(Patch2D& p) {
  double sum = 0;

  p.map_all(N, x, results);
  for (int i = 0; i < N; i++)
    sum += w[i]*mag(results[i]);

  return sum;
}
```

The loop in `map_all` can execute quickly because `map` can now be expanded. The overhead of the call to `map_all` is negligible because the routine is doing a relatively large amount of work. Higher level code can still be written in terms of the base class `Patch2D` because `map_all` is virtual. Furthermore, the `Vector2D` addition and scalar multiplication can also be expanded. This gives an optimizer or vectorizer all the information it needs (up to aliasing) to generate good code. An additional benefit is that the loop in function A2 is now far simplier and can be optimized. The performance differences between function A1 and function A2 can be dramatic, we saw over a factor of 5 improvement in our quadratures between the two codes, keeping all other conditions constant.

A2 is slightly more complex than A1, primarily due to the fact that some piece of code has to take responsibility for managing `results`. In FastScat we have encapsulated this additional complexity in quadrature classes so that it is completely hidden from the user. Users of our quadrature classes only need to supply "all" versions of methods that are performance critical. For the surface classes, only 4 out of over 20 methods have "all" versions.

By adding some additional methods to our classes, we have kept the benefits of object-oriented programming without sacrificing performance. The moral is to use object-oriented techniques in all but the very small percentage of code that is executed often. Such code must be understandable by the optimizer, meaning that it should be short, and written in terms of fundamental types like int and double. Fortunately, the use of object-oriented techniques allows us to structure the code into easily understood and fast *computational kernels*. The next section discusses this approach further.

3.2 Computational Kernels

FastScat does a significant amount of linear algebra, which is handled by the `Array` and `Matrix` classes. These classes call LAPACK [11] and the BLAS [12, 13] to perform the actual operations. LAPACK, a descendant of LINPACK and EISPACK, is intended to be highly portable and execute efficiently on a large range of target machines. The BLAS is a set of basic linear algebra subprograms, such as matrix-array (vector) multiplication, that are hand tuned to each machine. For example, on a Cray, the BLAS is written to take maximum advantage of the machine's vector units. A good implementation of the BLAS on a scalar machine would ensure that code and data are cached most efficiently and that the execution of the floating point and integer units is overlapped as much as possible.†

The actual implementation of the Array class is simple:

```
class Array {
public:
  complex dot(Array& b) {
    ZDOTU(&length, data, &stride,
         b.data, &b.stride);
  }
  ...
private:
  complex* data;
  int length, stride;
}
```

The routine ZDOTU is just a BLAS call that does a double precision complex dot product. The stride parameters tell how many elements to skip between consecutive array indices. Note that because dot can be inlined, the users of Array are effectively using the BLAS directly. This illustrates a useful technique: place C++ "wrappers" around high quality libraries implemented in other languages. The libraries then become C++ objects that can be used like any other object.

By carefully isolating the critical code in an application, the performance of an object-oriented program can be made as good as the best programs written in traditional languages. One additional benefit is that the object-oriented code is very portable. Only the kernels might need modification for a particular architecture.

3.3 C++ Limitations

Despite its rich set of features, C++ does have limitations. One that we found particularly frustrating is the lack of multimethods [14, 15], a generalization of virtual methods. A virtual method dynamically dispatches to code, which is selected based on the type of its first argument (this). A multimethod can dispatch on the types of many arguments. Consider a Tensor class that has descendants for Scalars, Vectors, Second-Rank Tensors, an so on, for which we want to define a set of arithmetic operations. The base class Tensor has a virtual method mul(Tensor) that must be defined by each derived class. The problem is in the implementation of mul in the derived classes:

```
Rank2::mul(Tensor& t) {
  select (t.is_a()) {
  case scalar :     // do
Rank2*Scalar
  case vector :     // do
Rank2*Vector
  case rank2  :     // do Rank2*Rank2
  // make higher rank class do the work
  default     : return t.mul(*this); }
}
```

This code is ugly, it cannot be inlined, and using is_a methods to return a type tag is a poor practice. The code is also difficult to maintain, because a class of a given rank must be a friend to all classes having a lower rank (scalar is rank 0, vector is rank 1, etc.). With multimethods, the solution is much cleaner and potentially more efficient because each method is responsible for only one kind of multiplication, for example, Scalar*Vector. Other solutions are possible in C++, but they are all similar in nature and suffer from the problems mentioned. This type of construction arises often in mathematics and it is unfortunate it does not have a clear expression in C++.

A second limitation of C++ is its lack of automatic memory management. Of course, any sort of memory management scheme can be implemented in C++, but we have found that a significant amount of effort goes into designing storage management solutions for various classes, and finding memory leaks. It is common for a C++ program to have several different storage management schemes. For example, in FastScat we use a reference counting technique [15] for the Array class to eliminate unnecessary copying of large objects, and an ownership-based scheme in the Composite_Surface class for patches. Several of the methods in a class (constructors, the destructor, and the assignment operator) must be concerned with memory management. The problem with multiple schemes and methods is that memory management must always be on the mind of the programmer and is a distraction from solving the problem of interest. We believe that some sort of default memory management, which can

† The array and matrix class were originally implemented entirely in C++. The implementation used the standard C convention that the rightmost index varies the fastest. When we switched over to the BLAS, we converted the internal storage format. Although this was a major data representation change, not a single line of code outside the matrix class had to be changed.

be overridden when necessary, would be beneficial.

Finally, C++ tools are still immature. Some vendors have been slow to implement language features, such as templates. Also, the lack of exception handling in most implementations makes error handling clumsy. These problems should disappear with time.

4 CONCLUDING REMARKS

Object-oriented programming is not without costs. We have noticed that it takes more time to design an object-oriented program than a procedural one, which is consistent with some estimates that up to 40% of the effort required to write an object-oriented program goes into the design phase. Also, when object-oriented languages are used in an overly procedural fashion (which is quite easy to do in C++), the benefits of the methodology are lost and the resulting code is often worse than a traditional program. This is similar to an effect noticed when Ada was first introduced. Many programmers were quickly retrained in the Ada syntax but not its design philosophy. Of course, the payback to putting more effort in the design, is in reduced debugging time and easier maintainability/extensibility.

The use of object-oriented languages for numerical applications is being hampered by the fact that object-oriented languages are not Fortran. Fortran is still the language of choice for a majority of people doing computational science, particularly on supercomputers. There are a number of reasons for this:

1. Supercomputer Fortran compilers typically vectorize code better than other compilers.
2. Fortran is widely understood.
3. A great deal of Fortran code exists.
4. There is a built-in resistance to change.

In order for object-oriented design and programming to make serious inroads in computational science, scientists and programmers are going to have to see some obvious benefits. We think the most convincing argument will come from the extensibility of object-oriented programs. If a computational scientist sees a group getting good results quickly, by virtue of being able to easily change their programs, the scientist will naturally become interested in the programming techniques.

In summary, we based the top-level design of FastScat on the physics of scattering. This lead to a flexible code that is easy to maintain and extend, and yet does not necessarily sacrifice efficiency. The fundamental calculations are performed by computational kernels such as the BLAS and a small set of hand-tuned methods in the quadrature classes. The high-level classes simply orchestrate the operation of the kernels. In the future, we plan to extend FastScat to handle more complex scattering problems and to port the code to massively parallel machines and to vector machines such as the Cray.

REFERENCES

[1] R. F. Harrington, *Field Computation by Moment Methods.* New York: Macmillan, 1968.

[2] R. Coifman, V. Rokhlin, and S. Wandzura, "The fast multipole method: A pedestrian prescription," *IEEE Antennas Propagation Soc Magazine,* vol. 35, pp. 7–12, 1993.

[3] V. Rokhlin, "Solution of acoustic scattering problems by means of second kind integral equations," *Wave Motion,* vol. 5, pp. 257–272, 1983.

[4] V. Rokhlin, "Rapid solution of integral equations of scattering theory in two dimensions," *J. Comput. Phys.* vol. 86, pp. 414–439, 1990.

[5] V. Rokhlin, *Diagonal Form of Translation Operators for the Helmholtz Equation in Three Dimensions.* Technical Report YALEU/DCS/RR-894, Yale University, Department of Computer Science, March 1992.

[6] B. Meyer, *Object-Oriented Software Construction.* New York: Prentice Hall, 1988.

[7] G. Booch, *Software Engineering with Ada.* Menlo Park, CA: Benjamin/Cummings, 1983.

[8] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison-Wesley, 1983.

[9] S. Keene, *Object-Oriented Programming in Common Lisp.* Reading, MA: Addison-Wesley, 1988.

[10] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual.* Reading, MA: Addison-Wesley, 1990.

[11] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostouchov, and D. Sorensen, *LAPACK User's Guide.* Philadelphia: Society for Industrial and Applied Mathematics, 1992.

[12] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, "Algorithm 679: A Set of Level 3 Basic

Linear Algebra Subprograms,'' *ACM Transact. Math. Software*, vol. 16, pp. 18–28, 1990.

[13] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ''Algorithm 656: An extended set of Fortran basic linear algebra subprograms,'' *ACM Transact. Math. Software*, vol. 14, pp. 18–32, 1988.

[14] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay, *OOPSLA Conference Proceedings*. Reading, MA: Addison-Wesley, 1991.

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