OBJECT CLASSES FOR NUMERICAL ANALYSIS

CRAIG C. DOUGLAS*, DAVID A. GEORGE*, AND MICHAEL E. HENDERSON*

Abstract. The current state of software development for numerical simulation is unnecessarily cumbersome and time consuming. The language numerical analysts use to communicate ideas with each other is fairly standard, even across language barriers. There have been successful efforts in computer aided designing (CAD) and visualization to make software systems which are designed so that tasks are performed in a way similar to the way they are commonly described. This is sufficient to us to indicate that if the interface between numerical subroutines and software were designed along similar lines, software development would be easier, quicker, more flexible, and could reuse more of previous development efforts.

We present some examples of what the design of such a numerical analysis object system might be. The examples use a prototype C++ implementation.

1. Introduction. In this paper we try to make the case that a high level, object-oriented design for the flow of data between the computational pieces of a numerical simulation would reduce the time and effort required to build new applications. This paper is not a description of a completed object-oriented numerical analysis object library. Rather, it is about what the design ought to look like. We provide examples from a system that works, but is in its infancy.

In §2, we discuss the current state of simulation development. In §3, we discuss what is possible today that would enhance simulation development and re-use of tools. In §4, we discuss a set of numerical analysis objects, currently defined using C++ classes. In §5, we provide some examples.

2. Current simulation development. There is a clear need for numerical simulation. This is due partly to the time and expense required to build and test prototypes and partly because there are many questions which cannot be answered by physical testing (e.g., where instrumentation is intrusive, scales are too small, or it must be done in space).

The capabilities for performing numerical simulation have improved tremendously in recent decades. However, most improvements have been aimed at increasing the size of problems and the complexity of the geometry and physics which can be simulated. Not enough has been done to shorten the process or to decrease the cost of creating and validating a numerical simulation.

Suppose we have a problem that can be solved best by numerical simulation. There are several

* IBM Research Division, T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598–0218, USA.

options for doing this:

- Find a package.
- Use several packages sequentially.
- Use a subroutine library in parts of our application.
- Write the entire application from scratch.

Too often, the latter two options are the only choices.

There are many application areas where simulation packages exist (e.g., structural mechanics, gas or oil flow in a pipeline network, or heat conduction). If our problem is in one of these categories, we can buy a package which includes geometry specification, mesh generation, tables of material properties, and visualization tools. While the simulation package does the bulk of the work, this may still require some amount of work on our part. Typically, the problem must be reformulated slightly or the input format to the package is different from what we already use, and so must be converted before or after the package is used.

Some simulations involve multiple stages in an entire process. Hence, it may be possible to use a sequence of packages. Suppose we are designing a new widget. We might start by drawing the proposed widget using a commercial CAD package. This provides the input to a mesh generator which produces an unstructured mesh. The mesh is used by a program to simulate the casting of the widget and outputs the residual stress in the widget. The computed residual stresses are then passed to a routine which simulates the loading of the widget in actual use. To do a simulation in this manner involves matching the output format of one package to the input format of another.

If the previous two approaches are not applicable, creating a new code using existing subroutine libraries may be possible. For example, the use of libraries to manipulate linear systems of equations and to produce graphics output is common. There are also repositories like *netlib* [6] and *MGNet* [7] where various research codes are available via anonymous ftp (NIST recently announced that it is making it's "Guide to Available Mathematical Software" available, see [2] [3]). For example, there are codes for mesh generation, ordinary differential equation (ODE) integration, and for solving boundary value problems. The user is responsible for getting data into the form required by these routines, which in many cases requires converting data from one format to another. This may have to be done in memory or by writing new disk files. Finally, the user must write (and debug) code that calls the library.

The final approach is to code the application from scratch. This occasionally offers better performance and avoids issues of ownership, but requires a larger investment of time, effort, and expertise.

Upon completion, these packages may become commercial codes, but often are used only in-house.

Obviously just knowing the equations and methods that are appropriate for a given problem is only a small part of performing a numerical simulation. Despite progress, it is usually much more difficult to code and debug an application than to design it. This time is not spent identifying and correcting problems with the algorithms, but with implementing and debugging data conversions and specifying parameters to various parts of the application code.

A typical implementation of an algorithm focuses primarily on data structures, input, and output. This is natural, since novel data structures are often the key to advanced algorithms. This approach greatly limits the usefulness of implementations and places a heavy burden on a developer who tries to make the code generally useful. A properly defined object-oriented interface between implementations of numerical algorithms could solve many of these problems. The main feature of the design is a shift in focus from the way data is stored to how it is accessed and modified.

In §§4–5, we describe what this interface could look like. The field of numerical simulation is moving in this direction already. Consider the emergence of standard file formats and subroutine interfaces (e.g., the BLAS [4] [5] [8]). However, at present this is piecemeal and slow. If this paper does nothing else we hope that it focuses attention on the problems which occur in piecing together simulations from existing software.

3. Possibilities. The approaches of building subroutine libraries for common tasks and building application packages for large classes of problems are well established. Further, they are chosen for particular reasons. In this section, we investigate issues that suggest that different approaches might be more successful.

The use of software is fundamentally a matter of describing the inputs on which the software acts, and the output which it is to create. A diverse community of researchers and users must agree on common ways of exchanging data. There is already some consensus about how this should be done, as evidenced by the language numerical analysts use to communicate. For example, a fluid dynamics simulation might be described in the following way:

- The geometry:
  - Two dimensional converging nozzle.
- The equations:
  - Interior: steady, incompressible, irrotational, inviscid, two dimensional fluid flow.
  - Top and bottom: no flow boundary conditions.
  - Left edge: parallel in flow, given velocity.

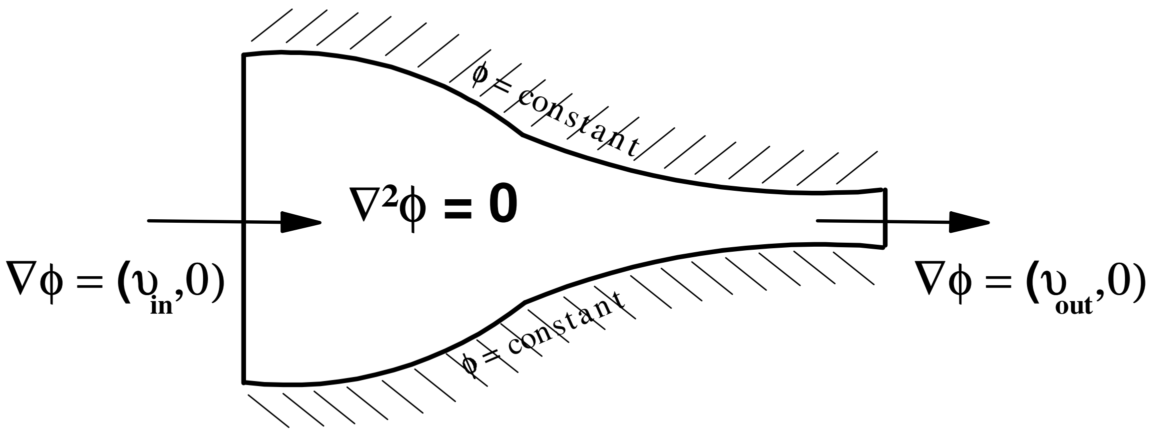

FIG. 1. *Steady, inviscid, irrotational fluid flow in a converging nozzle*

  - Right edge: parallel out flow, given velocity.
- The solution method:
  - Introduce a velocity potential.
  - Use a conformally mapped grid.
  - Discretize the resulting Laplace's equation using centered finite differences.
  - Solve the linear system using multigrid.

There are many assumptions here. We are not implying that programs must be written in this form or that all algorithmic issues be resolved in such a cursory form. Rather, there are concepts which occur naturally and are common across many different applications. These include:

- Regions.
- A discrete representation (mesh) of the regions.
- Functions on the mesh.
- Operator equations (interior and boundary operators).
- A discrete representation of the operators.
- A set of subproblems that must be solved to determine the function which satisfies the operator equation.

Given that an object-oriented interface between software components for numerical simulation should eventually reduce the programming effort required to build a simulation, the existence of a common language for describing the algorithms indicates that a design based on the concepts of the language has a good chance of being acceptable to those developing and using numerical algorithms.

Given that a design can be specified, can it be implemented in a usable way? There are at least two areas where such implementations exist and have been successful.

CAD packages are data bases of geometric objects coupled with methods for combining and manipulating those objects. Operations in CAD systems have been designed to correspond with the way that physical objects are described. For example, objects are assembled from sub-objects, which can be rotated, translated, and combined. Reference points and lines of symmetry can be described in terms of intersections of various other objects. Objects can be trimmed by removing sub-objects inside them.

Visualization packages have also been organized around a descriptive set of objects. Most have a basic object which is a function on a mesh. Additionally, there are glyphs, text, streamlines, and streaklines. A user constructs a visualization in a way that is similar to the graphics actions wanted. For example:

- Put a sphere at this point in the flow and track it with the flow.
- Draw a surface of constant pressure.

Since they deal with the output of numerical simulation, these visualization packages have a lot of features that are the same as those needed in numerical simulation. To indicate this correspondence, we consider a simple example which computes the result of applying a discrete Laplacian to a function on a rectangular mesh.

Figure 2 shows a *graphical program* from a visualization package. In the example, a scalar function is created over a square, regular mesh. Then the function is drawn, and the Laplacian (div grad) of the function are drawn. The basic steps are:

- Construct a grid.
- Define a function on the grid.
- Compute the gradient of the function.
- Compute the divergence of the gradient.
- Draw the function and its Laplacian (rubbersheet is a way of drawing functions in this package).

There are many things in this example that are analogous to the pieces used to develop a simulation, but some are simply inadequate. Notice that the grid, function, and operators are basic objects. We will augment these basic objects with the function that is necessary to write and solve partial differential equations. Additional information is required when describing equations on meshes or

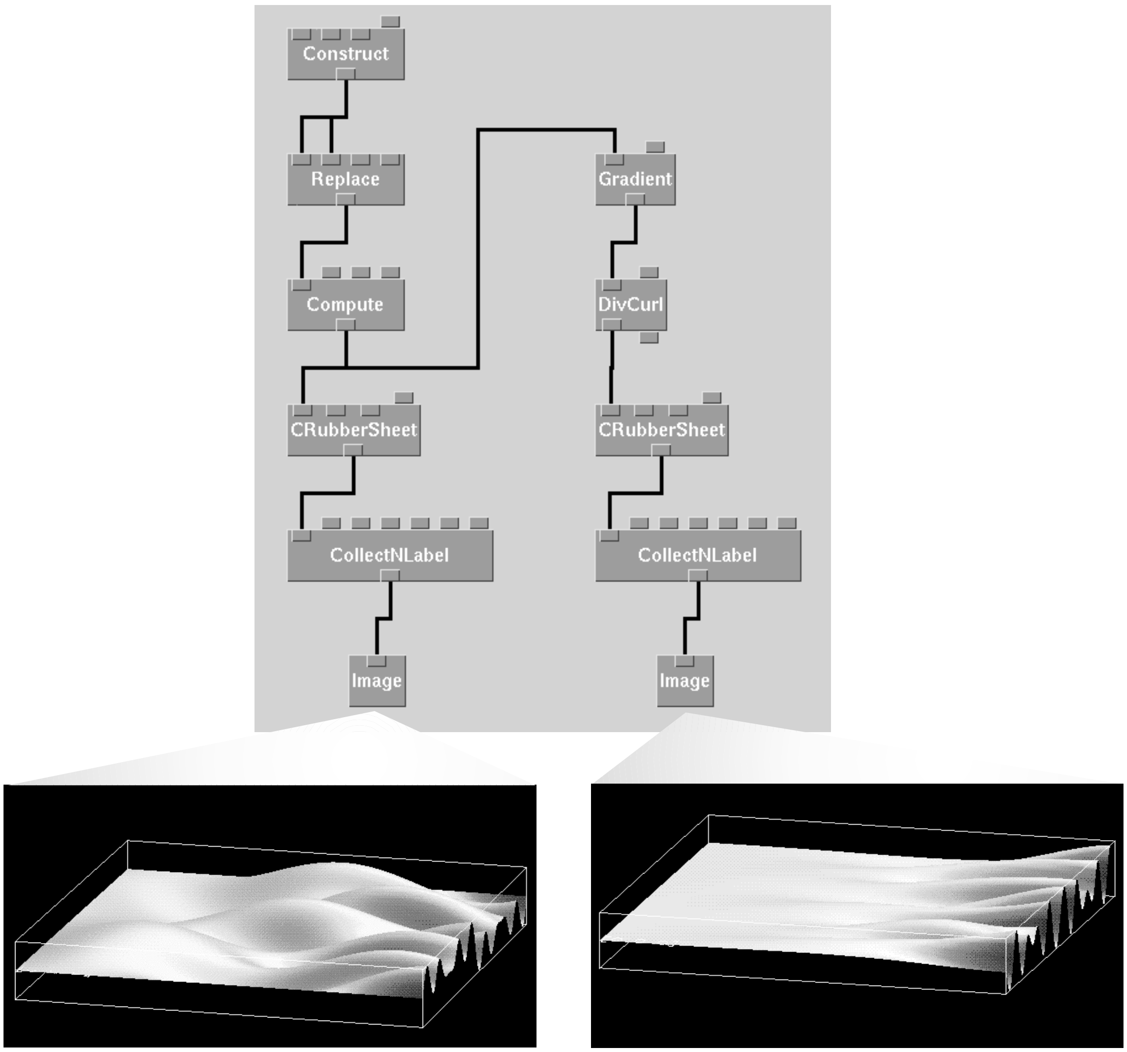

FIG. 2. *An example of a visual program from IBM's Visualization Data Explorer. The program creates a rectangular grid (Construct), defines a function on the mesh (Replace/Compute), then plots f(x,y)=*$\sin(\pi x \sin(\pi y))\cos(\pi x \cos(\pi y))$ *and its Laplacian (Gradient, DivCurl, CRubberSheet and CollectNLabel).*

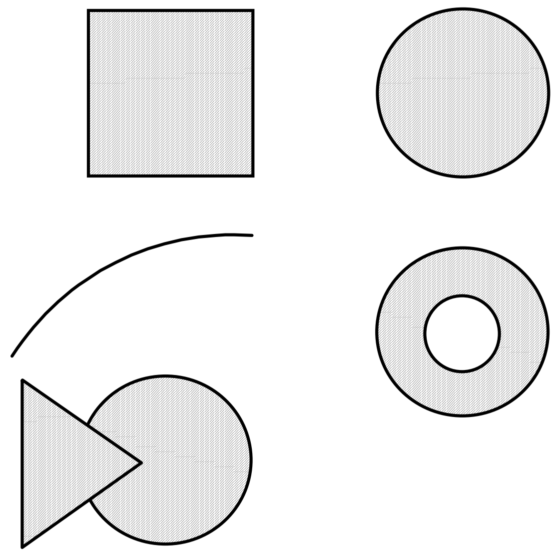

FIG. 3. *Examples of regions, which are a set of subsets of an n-dimensional space. Regions may be composed of several overlapping parts, and may have interior boundaries.*

solving operator equations, but is not necessary for visualization.

4. Numerical Analysis Objects. The three objects *regions*, *functions*, and *operators* are the fundamental object classes. We describe these basic objects and give some examples below. While these are sufficient to describe a simulation, major subclasses must be specified and implemented, and utilities for performing basic tasks with these objects must exist before they are useful. We have made a preliminary implementation of the base objects in `C++`, and are developing sub-classes and utilities. The purpose of this implementation is to verify that the base member functions and operations are sufficient. A finished system that can be used for real problems will be written after we have some experience using the preliminary implementation. Below we describe the fundamental object classes, and some of the subclasses which we have implemented.

4.1. Regions. A *region* is simply a subset of some n-dimensional space. For example,

- the interior of a circle,
- the circle, or
- the region bounded by four curves

can all be represented as region objects. Figure 3 shows some examples of regions. *Region* objects will be developed into the more general *Manifold* object in §4.4.

A region can have a boundary. In this case, the pieces of the boundary are themselves lower dimensional regions. Figure 4 shows a two-dimensional region with a boundary which is made up of

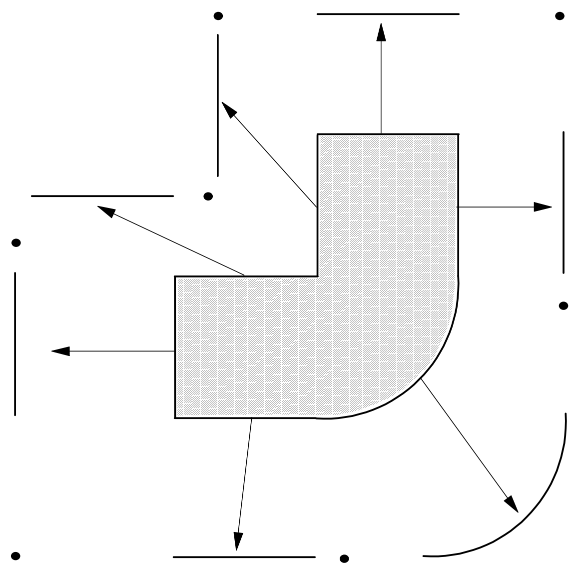

FIG. 4. *Example of a region with boundaries. The boundary is represented by a list of lower dimensional regions, which is this case includes both curves and points.*

one and zero dimensional regions.

Regions can be made up of several sub-regions, possibly overlapping. They can also be discrete (cellular complexes), i.e., made up of element and point lists. Figure 5 shows a discrete region with nine points and eight elements.

4.2. Function objects. A *function* is a mapping from one region to another. For example,

$$\sin(x) : [0, 2\pi) \rightarrow [-1, 1].$$

A function has a domain region and a range region. Additionly, there may be a norm or inner product associated with a set of functions. The information about domain, range, norm, or inner product is associated into a *functionSpace* object. Functions in the same functionSpace can be added. Functions with compatible functionSpaces can be composed.

The basic operation a function provides is *evaluation*. Evaluating a function at a given a point in the domain region yields a point in the range region.

We have implemented three types of functions (ExpressionFunctions, TableFunctions, and ExternalFunctions) and several ways of combining functions. These are all derived from the base class Functions, but provide their own constructors and evaluate member functions.

4.2.1. Expression Functions. ExpressionFunctions take a set of strings (one for each dimension of the range region) which are expressions involving the coordinates of a point in the domain

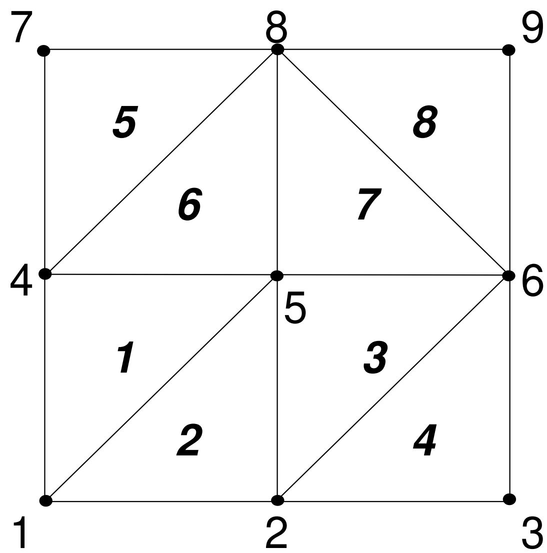

FIG. 5. *An example of a discrete region. A discrete region consists of a list of points (1-9), a list of elements (1-8) which are polygons (or polyhedra) whose vertices are in the point list, and a list of the neighboring element for each face of each element.*

region. For example, if the domain is $\mathbb{R}^3$, and the range is $\mathbb{R}$, the program

```
Region *threeSpace=new RealEuclideanNSpace(3);
Region *R=new RealEuclideanNSpace(1);
FunctionSpace *FS=new FunctionSpace(threeSpace,R);
Function *f=new ExpressionFunction(FS,"sin(x0)+x1**2/x2");
```

creates domain and range regions, a functionSpace, and an expressionFunctions containing a *compiled* version of the source string that is executed when the evaluate function is invoked.

4.2.2. Table Functions. TableFunctions require a discrete region as their domain. One point in the range is stored for each point in the domain (cf., the way functions on finite difference meshes are stored). A natural extension is an interpolatedFunction, which is constructed from a tableFunction and a function that interpolates on the elements.

4.2.3. External Functions. ExternalFunctions (or *user call backs*) are constructed by providing the address of a routine, which is saved. It is invoked with the proper arguments by the evaluate member function.

4.2.4. Composite Functions. In addition to these three basic types of functions, classes have been implemented which allow composite functions to be created, such as the products

$$F(x,y,z)=(a(x),b(y),c(z)) \qquad \text{or} \qquad G(x)=(f(x),g(x)).$$

These store a pointer to a list of functions. The evaluate for the productFunction invokes the evaluate for each function in the list using the natural ordering, then constructs the vector containing the result.

There are also classes for the composition of functions and for the sum, difference, product, ratio, and scalar multiples of functions. These are not efficient, but allow dissimilar functions to be combined. Individual classes can use overrides to provide this functionality efficiently. For example, the sum of two Functions would be handled by a *sumOfTwoFunctions* sub-class, which just stores pointers to two Functions. For evaluation, it sums the result. To sum two functions within a particular class of functions, the class might construct a new instance and compute the appropriate values (a tableFunction would store a list of sums). The user may also wish to use the sumOfFunctions class if storage is a concern.

4.3. Operator objects. Operators are similar to Functions in the sense that they have a domain, range, and possibly a norm. Operators are associated with OperatorSpaces instead of FunctionSpaces. The main difference between functions and operators is that the domain and range of an operator are functions. Thus, a derivative operator might act on a scalar function of two scalars and return a vector function of two scalars. (The gradient is an example of such an operator.) Operators have a member function called *apply* (to distinguish operators from functions, which are evaluated).

We have implemented three classes of operators (MatrixOperators, FiniteDifferenceOperators, and FiniteElementOperators. Finite volume methods can either be implemented as a separate class of operators, or may be cast as finite element operators, see [1]) as well as some classes which are composites of operators.

4.3.1. Matrix Operators. MatrixOperators store a linear system in an internal format. The apply operation is a matrix–vector multiply. We have implemented two sub-classes (FullMatrices and BandedMatrices) and plan to eventually provide others. We provide simple constructors and member functions that allow entries in the matrix to be recovered and set, and for linear systems to be solved.

Matrices reveal an ambiguity in the commonly used language. A matrix is a linear function which maps one vector space into another. However, it is also sometimes considered to be a linear operator on linear functionals (e.g., mappings from vector spaces into the real numbers). We have

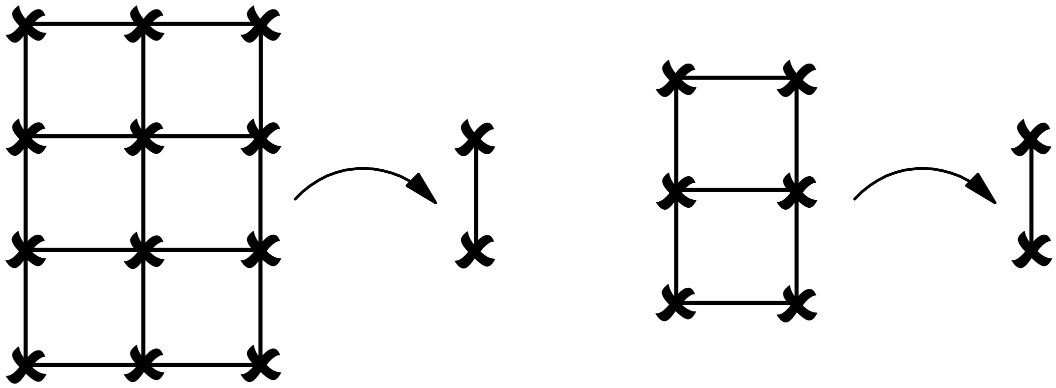

FIG. 6. *Examples of FiniteDifferenceOperators. Finite Difference Operators map one function on a discrete region into another. We show the domain (a discrete region) of the domain and range function spaces of the operator. The left hand example shows the second order Laplacian* ($E_x + E_x^{-1} + E_y + E_y^{-1} - 4I$)*. Only the interior points have values. The right hand example show an averaging operator* ($E_x^{1/2} + E_x^{-1/2} + E_y^{1/2} + E_y^{-1/2}$)*. Note that the domain of the range function space is not a subset of the domain of the domain function space.*

implemented them both as Operators (finite element and finite difference methods can be considered as a method for approximating differential operators as matrices) and as Functions.

4.3.2. Finite Difference Operators. FiniteDifferenceOperators (FDO) are a base class derived from the Operator class that provides a *rewriteAsMatrix* member function. They can only act on functions whose domains are discrete regions that have only square or cubical elements. We have implemented sub-classes with a general stencil representation, and with particular operators. The range of these operators is closely tied to the domain. The range function is a tableFunction defined on a domain which is missing those points at which the FiniteDifferenceOperator cannot be applied. The operator may also return values at points not in the domain (the average of neighboring grid points) (see Figure 6).

An example is the second order Laplacian. This is a sub-class of FDO's. The stencil coefficients are not stored since the apply function is appropriately overloaded. This class is necessary because the Laplacian is a coordinate-free operator. Which combination of derivatives in the coordinate directions constitutes the Laplacian requires some information from the domain of the function on which the Laplacian operates.

Another example of additional information which is required when writing differential equations is the periodic line segment $[0, 2\pi)$. This region is not geometrically periodic, but the distance between $2\pi - \epsilon$ and ϵ is 2ϵ, not $2\pi + \epsilon$. How to write coordinate-free operators and information about distance are provided by Manifold objects, which are a generalization of the region object.

4.3.3. Finite Element Operators. FiniteElementOperators (LFEO's) are implemented similarly to the finite difference operators. There is a base class that adds an assemble member function (the rewriteAsMatrix step) and a general way of storing a finite element operator. Instead of storing a set of stencil coefficients, the operator is represented as a linear combination of inner products of a set of basis functions. For example,

$$(Lf, \phi_i) = \sum \left[-Af_i(\nabla\phi_i, \nabla\phi_j) + \omega^2 f_i(\phi_i, \phi_j) - g_i(\phi_i, \phi_j) \right] .$$

4.4. Manifolds. The last object class we describe is the Manifold. The discussion of the Function and Operator objects relied on a Region object. For most purposes the idea of a piece of the line, plane, or three space is sufficient, but there are cases where a more general concept is necessary.

In many situations, multiple regions are desirable. This can simplify problem specifications and reduce a complicated domain to a set of simpler ones. One idea that appears frequently is to map a simple domain (e.g., a rectangle) into the region of interest (perhaps a converging nozzle). This is clearly being done in conformal mapping and also appears with multi-block and overlapping grid methods.

A Manifold is a set of maps (or Charts) that take some subset of a base space (e.g., a neighborhood of the origin in R^n) into a target space. The set of Charts is called an Atlas. If the Charts have some smoothness (i.e. the maps agree where neighborhoods overlap), then the Atlas defines a manifold.

We have used this basic approach to define a Manifold object, which we believe is sufficiently general to deal with most geometries and meshes. It is therefore somewhat complex, so we present it in stages.

The Manifold object can be queried to determine the base and target space dimensions and the number of Charts in the Atlas. A Chart can be applied (see Figure 7). Additionally, a point in the base space can be checked to determine if it is in the neighborhood corresponding to a particular Chart, and a point can be mapped between Charts. There is no stipulation that the Charts overlap over finite regions, so we also allow manifolds that have cusps and perhaps even jumps.

Those are the member functions which all Manifolds have. Additionally, a Manifold may have a boundary (i.e. it may not be open), it may be discrete, and it may have a differential structure.

The boundary of a Manifold is represented by a set of manifolds with the same target space, but

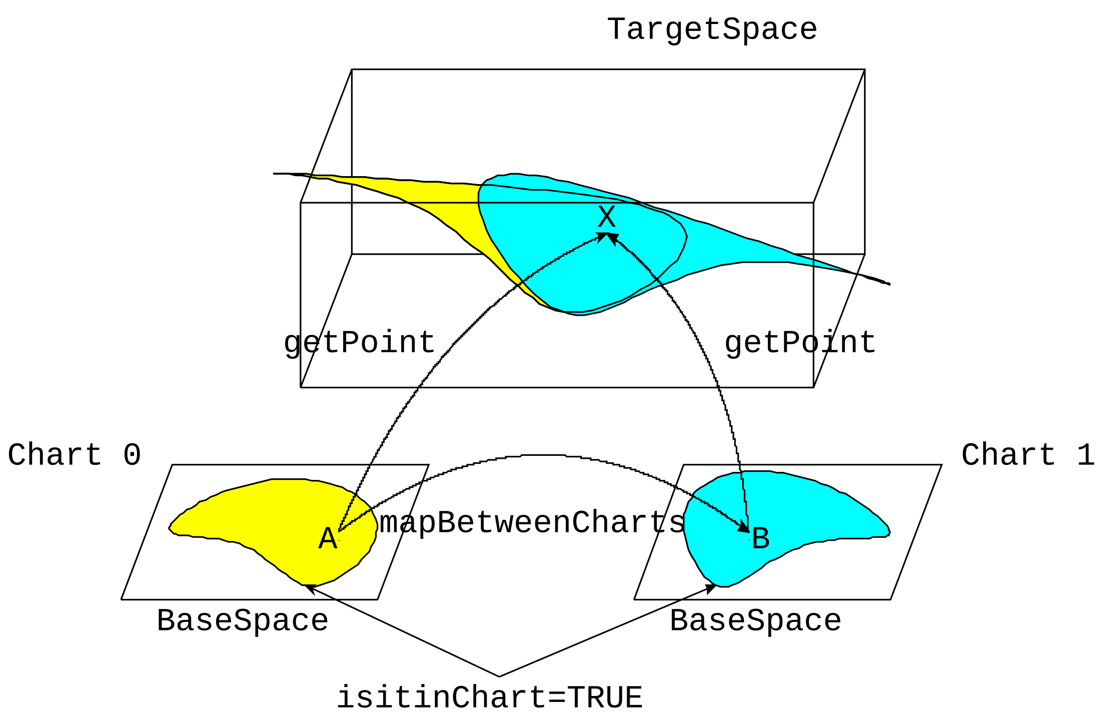

FIG. 7. *Common Manifold member functions. This figure illustrates the member functions that all Manifolds have. The example has a two dimensional base space and a three dimensional target space. There are additional member functions if the Manifold is discrete, has a boundary, or a differential structure.*

a base space of lower dimension. The Manifold can be queried to determine how many boundaries exist, and each of the pieces of the boundary can be extracted. (See Figure 4.)

A discrete Manifold has a list of points and a list of cells or elements. Each cell is a set of faces that are cells of one lower dimension. Thus, a cubic cell has faces which are squares. The vertices of the cells and faces are members of the point list. Additionally, the neighboring cell across a face can be determined. (See Figure 5.)

This is a very general definition of a grid, called a cellular complex. Finite element grids are essentially cellular complexes, but may omit information about neighboring cells. Finite difference grids are usually much simpler: the points, cells, and neighbors can be computed directly from an index.

The last feature of Manifolds, a differential structure, was alluded to in §4.3. A smooth manifold has a tangent space and normal space at each point. Coordinate-free differential operators (such as the Laplacian, gradient, and divergence) are defined in terms of a metric and a connection. These provide information about how the tangent and normal spaces change from point to point on the manifold. The metric represents the inner product of the tangent and normal vectors. The connection is the derivative of the tangent and normal vectors along coordinate directions. Many differential equations are expressed in coordinate-free form. Associating this differential structure with a manifold allows coordinate-free discrete operators to be defined.

In summary, a Manifold is a very general type of region. It consists of a set of regions with rather arbitrary parameterization. They can have boundaries, a differential structure, and can be discrete.

5. Examples. Now that we have defined the three basic object classes and sketched some of the sub-classes, we turn to the question of how these might be used in a realistic setting. We discuss two examples. The first shows how existing implementations of algorithms are modified to accept and produce these objects. The second indicates how algorithms can be implemented at a high level, perhaps with some sacrifice of performance for increased applicability.

5.1. Incorporating ODE integrators. A large class of ODE integrators work on problems of the form

$$\begin{cases} u' = f(u,t), \quad t \in [t_0, T], \\ u(t_0) = u_0. \end{cases}$$

There are many integrators readily available. The choice is based on personal choice and which class of problems are to be solved. They all have different ways of specifying input, but all require the function f, u_0, t_0, and T. All use user call back routines: some assume a particular name of the routine, others allow an address of an arbitrarily named one as an argument.

We show here how the input to and output from these different packages can be cast in terms of the objects described above and indicate how such a wrapper code would be written. The main advantage for these particular routines is that the function f can be supplied in forms that the routine does not know a priori and that the output is in a standard form.

Consider how this problem would be expressed in mathematical language:

> Let u_0 be a point in R^n and $f : R^n \times R \to R^n$. Find the function $u(t) : R \to R^n$, for $t \in [t_0, T]$, such that $u(t_0) = u_0$ and $u' = f(u, t)$.

We are calculating a function that maps an interval of the real line into R^n. The input is a point u_0 in R^n (a point on a Manifold), a function f (mapping a product of two Manifolds into a Manifold), and an interval of the real line $[t_0, T]$ (a Manifold with boundary).

We might have the prototype

```
Function *ODEIntegrator(Function *f, Manifold *timeInterval,
                                     Tuple *initialPoint);
```

(In our implementation *Tuple*'s are objects that store a number of coordinates.) There are restrictions on the various inputs that the `ODEIntegrator` should check. For example, the `timeInterval` Manifold must be a interval on the real line, the `initialPoint` must have the same dimension as the range of the function `f`, and so forth.

Since we are working with an existing routine (and should not change the way it evaluates the function), our input has to be converted to a form that can be passed to the integrator we have chosen. We chose a standard ODE package from netlib. This package requires two routines of a specified name as user call backs.

We must provide a function which evaluates f, so `ODEIntegrator` would store a pointer to f and provide an auxiliary subroutine that uses the stored pointer to call the *evaluate* member function of f. If the problem is stiff, the ODE solver can take advantage of having the derivative of f or can use differencing. The user passes a flag that indicates which method to use. Functions can be queried to determine if an *evaluateDerivative* exists, which may be the user's favorite difference scheme or a coded derivative. `ODEIntegrator` checks to see whether f has this property and provides a subroutine that evaluates the derivative.

The rest of `ODEIntegrator` consists of setting up an initial step size, work arrays, and initial values. A TableFunction is also created for the output, whose domain is an IrregularDiscreteManifold and whose range is R^n. An IrregularDiscreteManifold in one dimension is simply a list of points, that are the time values at which the integrator computes the values of u. The TableFunction allocates a list of values in R^n, one for each point in the IrregularDiscreteManifold. As the integration proceeds, we set the values of the t and u and return when $t = T$ is reached.

5.2. FiniteDifference solution of Laplace's equation. This is an example of how routines can make use of our sub-classes to implement a fairly general finite difference scheme. It is always possible to do this more efficiently, but we are aiming for a code that is easily debugged and modifiable, and that will work on as large a set of problems as possible.

The problem is stated:

> Let Ω be an open region of the plane with boundary $\delta\Omega$. Given functions $f : \Omega \to R$ and $g : \delta\Omega \to R$, find the function $u : \Omega \to R$ such that $\nabla^2 u = f$ in Ω and $u = g$ on $\delta\Omega$.

This routine is going to take a Manifold and two Functions and produce a Function. We might therefore have the prototype:

```
Function *solveLaplace( Manifold *Omega, Function *f, Function *g );
```

Finite difference schemes only work on particular types of discrete Manifolds, so `solveLaplace` needs to do is check to that the inputs are valid. The input discrete region must have a two dimensional base space, quadrilateral elements with four elements around a point, and the Functions f and g must have the proper domains and ranges.

The general flow of the program is:

- Check inputs for correctness.
- Construct an FDO that approximates the Laplacian.
- Construct an FDO that approximates the identity on the boundary.
- Rewrite each FDO as a Matrix.
- Rewrite the Functions f and g as tableFunctions F and G.
- Rewrite the tableFunctions F and G as linearFunctionals.
- Combine the interior linearMatrix and the boundary linearMatrix into one linearMatrix.
- Combine the interior linearFunctional and the boundary linearFunctional into one linear-Functional.
- Solve the linear system.

- Rewrite the resulting linearFunctional as a tableFunction and return.

The FDO's may be constructed by providing a stencil, by adding basic FDO's, or by using a coordinate-free FDO `Laplacian`. The step of writing tableFunctions as linearFunctionals is merely a matter of using the enumeration of points in the discrete Manifold to create a vector.

6. Conclusions. The advantages of a generally accepted, standardized interface between codes and implementations of algorithms for numerical simulation are obvious. We have presented one way of constructing such an interface. We are in the process of developing a trial implementation.

There are several concerns that have surfaced repeatedly. The first is efficiency. Researchers work very hard to create fast algorithms. If the overhead associated with such a system is too large, then the implementation is unacceptable. Some overhead is unavoidable. However, if the implementation is allowed to check the type of input, and to directly access (for example) the lists of points stored in tableFunctions, it is possible to make implementations which take advantage of special storage formats. At the same time code which works less efficiently can be included for input which in different forms.

Algorithms that are conceptually simple often require parameters, tolerances, and choices concerning algorithmic variants. A system that attempts to hide *all* of these complexities is also unacceptable. If different algorithms (with different numbers and types of parameters) are to be used interchangeably, either the set of parameters must be generalized so that all algorithms of a given class accept (but may not use) the same parameters, or the parameters must be given reasonable defaults, and a way provided for the user to override those defaults.

REFERENCES

[1] R. E. Bank and D. J. Rose, *Some error estimates for the box scheme*, SIAM J. Numer. Anal., 24 (1987), pp. 777–787.

[2] R. Boisvert, *Guide to available mathematical software*, in NA Digest, C. Moler, ed., vol. 94(11), March 1994.

[3] R. F. Boisvert, S. E. Howe, and D. K. Kahaner, *GAMS: A framework for the management of scientific software*, ACM Transactions on Mathematical Software, 11 (1985), pp. 313–355.

[4] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of Level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.

[6] J. J. Dongarra and E. Grosse, *Distribution of mathematical software via electronic mail*, Comm. ACM, 30 (1987), pp. 403–407.

[7] C. C. Douglas, *MGNet: a multigrid and domain decomposition network*, ACM SIGNUM Newsletter, 27 (1992), pp. 2–8.

[8] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, ACM Trans. Math. Soft., 5 (1979), pp. 308–323.