Vector-Bundle Classes Form Powerful Tool for Scientific Visualization

David M. Butler and Steve Bryson

Vector bundle theory provides an abstraction general enough to capture the common features of many scientific data formats

Object-oriented programming promises to improve programmer productivity by making software easier to reuse. Inheritance is often seen as the key to reuse, but it is essential to realize that inheritance leads to reuse only when the proper abstractions are known in advance. Inheritance is deductive. It proceeds from the general to the specific; the more general parent class must already exist for the child to inherit it. Unfortunately, software developers are usually inductive. They discover the general by programming several special cases. In this inductive approach, the object-oriented paradigm can actually lead to more work, not less, as existing code must be rewritten to "reuse" the abstractions as they are discovered.

In spite of the common experience we have just described, it is precisely because of the deductive nature of inheritance that object-oriented programming is well suited to the scientific domain. For at least four centuries, scientists have been developing and expressing knowledge in terms of mathematical abstractions. The proper abstractions for scientific data are known. We just have to use them. The first step in developing useful class libraries is thus to look at the mathematical formalism that science uses, rather than trying to generalize existing data formats. Abstract classes that capture the mathematical formalism form the parent classes from which format-specific representations can be derived.

Vector-bundle theory is a mathematical formalism that is general enough to provide a common ancestor for very many application-specific data formats. The vector-bundle data model uses the formalism to define several abstract data types, and then treats the various data formats and mesh types commonly found in scientific data processing as representations of these abstract data types. Visualization algorithms described and implemented in terms of the abstract data types can be used with any of the various specific representations.

The main ideas of the vector-bundle data model were introduced in a previous paper.[1] In this article, we focus on the mathematical structure of vector bundles and describe the object-oriented implementation of the model. Specifically, in the following section we describe the mathematical structure. Next, we describe an abstract class hierarchy that models vector bundles, and give an example of how the classes can be represented. Finally, we show how they can be used to construct visualization algorithms. [*For more on object-oriented programming, see the three feature articles in the Sep/Oct 1992 issue of* Computers in Physics.—*Ed.*]

Vector-Bundle Formalism

A vector bundle is a structure that formalizes the idea of having data at every point of some space, called the base space. The intuition is that the dataset at a point is an element of a vector space, while the base space containing the points may have a complicated structure. Vector bundles are already widely used in physics, in such topics as gauge theories in particle physics and in general relativity. Our objective in this article is to describe how this mathematical formalism is also useful in visualization. To do this, we first need to define some of the language of vector bundles.

Abstractly, a vector bundle is a collection (E, B, V, π), where E is called the total space, B is the base space, V is the vector space at each point of the base space, and π is

David M. Butler is a physicist and systems analyst with Limit Point Systems, Inc., 39807 Paseo Padre Pkwy., Fremont, CA 94538; e-mail: butler@sandia.llnl.gov. Steve Bryson is a computer scientist with the Applied Research Branch, Numerical Aerodynamic Simulation Systems Division, at NASA Ames Research Center, MS T045-1, Moffett Field, CA 94035; e-mail: bryson@nas.nasa.gov.

a projection that maps a point in E to a point in the base space B. Instead of (E, B, V, π), one usually just writes E for short. The vector bundle E has additional structure, which we will describe below, but first a particular example will help clarify these terms.

Consider the velocity field of a fluid flowing through some volume. The base space B is the volume containing the fluid. The set of all possible velocities of the fluid at a point is a vector space called the tangent space. In this case, V is the tangent space of a point in the volume B. E is the union of the tangent spaces over all the points in the volume. This union intuitively "bundles" the vector spaces together, hence the name vector bundle. The particular vector field that describes the actual velocity of the fluid is a function that assigns a vector $v(x) \in V$ to each point $x \in B$ in the volume. In vector-bundle language, the vector field $v(x)$ for all $x \in B$ is known as a *section* of the vector bundle E. The projection π is the function that assigns to each vector $v(x)$ the location x at which that vector is located. The dimension of the vector bundle is defined as the dimension of the vector space V.

In this example, V is the tangent space of B, and so has the same dimension as B. But in general, V can be any set with the structure of a vector space, and its dimension is independent of B. Acceptable vector spaces include the real n-dimensional vectors and tensors, as well as their complex and quaternionic generalizations. The definition of a vector bundle and its various operations applies for any vector space V. This is one of the reasons why an abstraction based on vector bundles is useful for many applications in scientific visualization.

The structure of manifolds. In scientific visualization, the base spaces of typical vector bundles are either physical objects or the space surrounding such objects. These spaces can have complicated structure; practical problems demand base spaces with holes or disjoint pieces. The appropriate mathematical structure for describing these base spaces is called a manifold.

Informally, a manifold is a space that is constructed by mathematically gluing together patches of Euclidean space. The procedure is analogous to using papier-mâché to construct a mask, with holes for the eyes and mouth. One cuts many rectangular strips of paper, then overlaps and glues them together into the desired shape, leaving holes as needed. The mathematical process is similar, using n-dimensional Euclidean space instead of two-dimensional paper, and coordinate transformations instead of glue. Spaces with holes, disjoint pieces, and certain other complicated structures can be constructed in this way.

More formally, an n-dimensional manifold consists of two objects: a topological space M, and a collection of coordinate charts (U_i, ϕ_i), where U_i is a neighborhood of M, and ϕ_i is a one-to-one map from U_i to a neighborhood of n-dimensional Euclidean space. By one-to-one, we mean that ϕ_i maps each point on the neighborhood U_i to a point in the Euclidean neighborhood so that two different points in U_i never map to the same point in the Euclidean space. Where the U_i's overlap, coordinate transformations are defined that take the coordinates in one chart and transform them to the coordinates in the other chart. Say that U_1 and U_2 are two intersecting coordinate neighborhoods. If ϕ_1 and ϕ_2 are the coordinate maps, then on the

overlap the composition of ϕ_2 with the inverse of ϕ_1, written $\phi_2 \circ \phi_1^{-1}$, will take a coordinate in U_1's chart and give the coordinates of the same point on the manifold in U_2's chart. The coordinate transformation defined by $\phi_2 \circ \phi_1^{-1}$ is referred to as a transition function from chart 1 to chart 2. Similarly, $\phi_1 \circ \phi_2^{-1}$ is a transition function from chart 2 to chart 1. Note that two coordinates related by a transition function both refer to a single point in the manifold. In this sense, they have been glued together.

This description of manifolds as local charts that are pieced together is valuable for several reasons. The most fundamental reason is that many topological spaces cannot be given coordinates that cover the whole space.

A typical example is the two-dimensional surface of a sphere like the Earth. Any assignment of pairs of coordinates, such as longitude and latitude, will necessarily have a place where the coordinates are not well defined. For instance, longitude is not well defined at the Earth's north and south poles. In order to give good coordinates to every point of the sphere, it is necessary to define at least two coordinate charts (U_1, ϕ_1) and (U_2, ϕ_2). There are many ways of defining these charts, but we need only the general picture, as shown in Fig. 1. Take U_1 to be the upper 2/3 of the sphere, and take U_2 to be the lower 2/3. The coordinate functions map each of these neighborhoods to some portion of R^2. These coordinate neighborhoods overlap in the middle third of the sphere. Any point on the sphere can be given two coordinates by use of these charts.

There are several pragmatic reasons for introducing the concept of manifold into numerical calculations, even though the topology of common problems can usually be handled by other means. As an example, it may be numerically advantageous to use a particular coordinate system, even though it is singular at certain points. For instance, spherical coordinates for three-dimensional Euclidean space are not one-to-one along the z-axis. This singularity is an artifact of the coordinates; Euclidean space is not in any way peculiar along the z-axis. Near such singularities, numerical roundoff errors can introduce difficulties into computations. Using the manifold concept, one can provide two different systems of spherical coordinates, with the same origin but the z axes at right angles to one another. Then the choice of which coordinate system to use can be determined by the angle of the desired point with the z-axis. While the example discussed here is rather simple, a similar problem arises in numerical relativity problems, where purely coordinate singularities can cause calculations to fail completely.

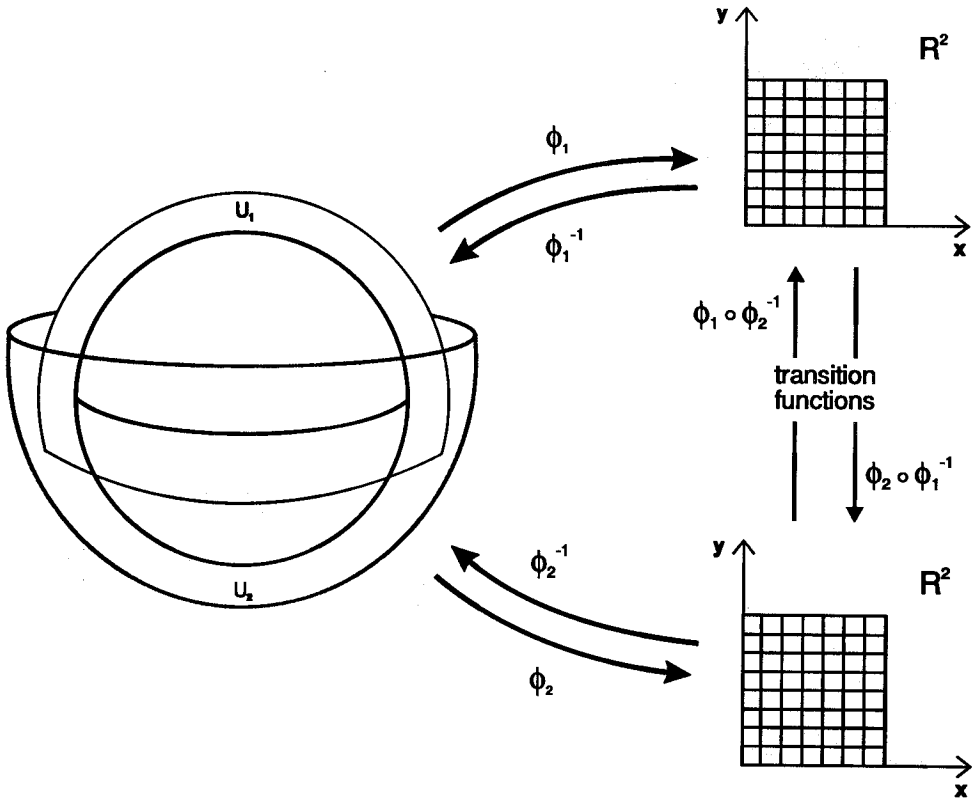

Fig. 1: Two coordinate charts are required to cover the sphere with one-to-one continuous coordinates.

As another example of manifolds in numerical computation, multiple overlapping computational grids are often used in fluid-dynamics calculations. For instance, in Fig. 2 the space around the airplane is covered by multiple grids. Each point of the space filled by the grid has computational coordinates, typically defined by floating-point conversions of the indices of the array that stores the grid. These computational coordinates are precisely analogous to coordinate charts of a manifold. Where the numerical grids overlap, one often needs to change from one set of computational coordinates to another. This is precisely what is done by the transition functions of a manifold.

The computation of the streamlines of a flow, as shown in Fig. 3, is a common example of how the transition functions may enter a calculation. The fluid-

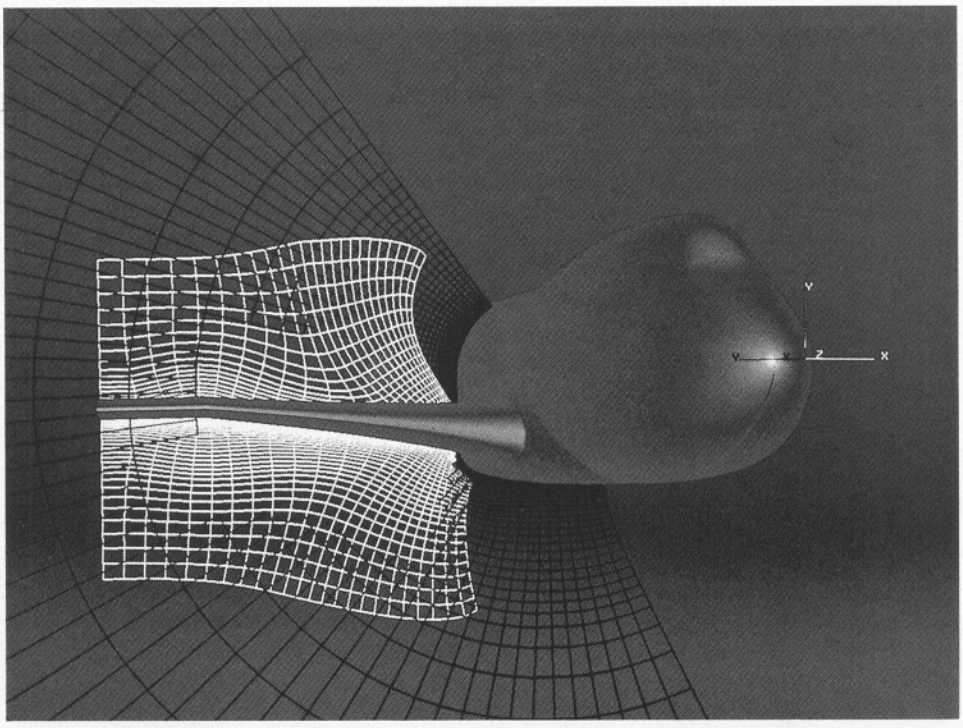

Fig. 2: Multiple overlapping grids cover space around an airplane for a fluid-dynamics computation.

velocity vector is defined at each point of each grid. Computing a streamline of the flow is the same as integrating this vector field given an initial point in the flow. This process is sketched in Fig. 4. These streamlines will typically cross grid boundaries. Historically, the integration has taken place in physical, not computational, coordinates. Real-time visualization of these streamlines, such as that in the virtual wind tunnel at NASA Ames, requires that the integration take place in computational coordinates. Then, as a streamline crosses from one grid to another, the computational coordinates in the previous grid must be converted into the computational coordinates of the new grid. This could be done by converting the old computational coordinates into physical coordinates and then into the new computational coordinates, but this tactic impedes real-time perfor-

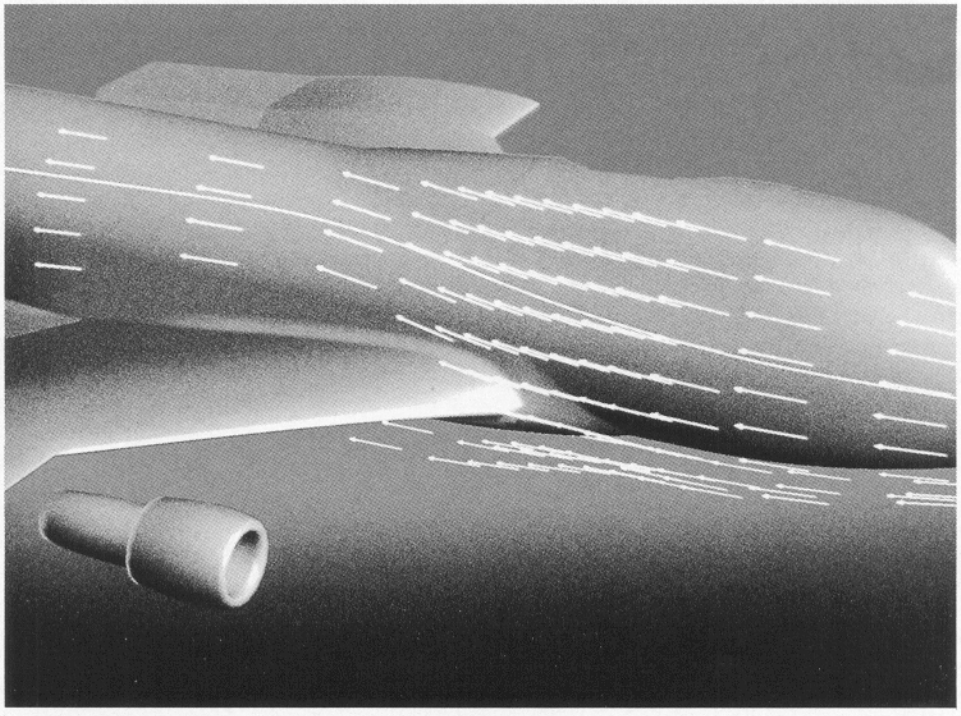

Fig. 3: Visualization reveals streamline in the flow around an airplane.

mance. The concept of the transition function of a manifold can be implemented as a look-up table, which takes the old computational coordinates and provides the new ones.

Before leaving the subject of manifolds, we will return to the formal definition. There is a natural hierarchy of structure in the definition of a manifold. Manifolds are topological spaces with coordinates. Topological spaces are point sets with neighborhoods or connectivity between the points defined. Finally, one is left with point sets, which are just sets of named, but otherwise featureless, objects. This mathematical hierarchy provides a detailed design guide for the object-oriented class hierarchy we will develop below.

The structure of vector bundles. In the same way that manifolds are constructed by gluing together patches of Euclidean space, vector bundles are constructed by gluing together vector-valued functions defined on patches of Euclidean space. If manifold construction is analogous to papier-mâché, then bundle construction is analogous to "carpet"-mâché, with the fibers in the carpet thought of as (one-dimensional) vector spaces. "Carpet"-mâché requires a fancier kind of glue, one that glues carpet backing to carpet backing and fiber to fiber. Without this special glue, we do not get a curved piece of carpet as a result of the gluing; we just get a matted mess. Similarly, vector bundles require more complicated transition functions that transform the coordinates of the base space and separately transform the components of the vector space at each point.

We now describe the remaining structure of a vector bundle, which we postponed previously. A vector bundle has a collection of coordinate charts (U_i, ϕ_i), where U_i is a neighborhood of B, the base space, and ϕ_i is a map from the collection of vector spaces attached to the points of U_i, written $\pi^{-1}(U_i)$, to the Cartesian product $U_i \times V$, where V is the vector space of the bundle. The map ϕ_i further satisfies the condition that it map the vector space attached at point x to $x \times V$; it does not somehow scramble up the attachment of vector spaces to points.

The collection of charts covers the base space, and, where the neighborhoods U_i overlap, we have bundle transition functions defined. The neighborhoods U_i are not necessarily the same neighborhoods used to define the manifold structure of the base space, and the bundle transition functions are different from the manifold transition functions. The transition functions are once again defined as compositions of the charts and chart inverses, and the charts must be defined so that every transition function has the form $(x, v) \rightarrow (x, g(v))$,

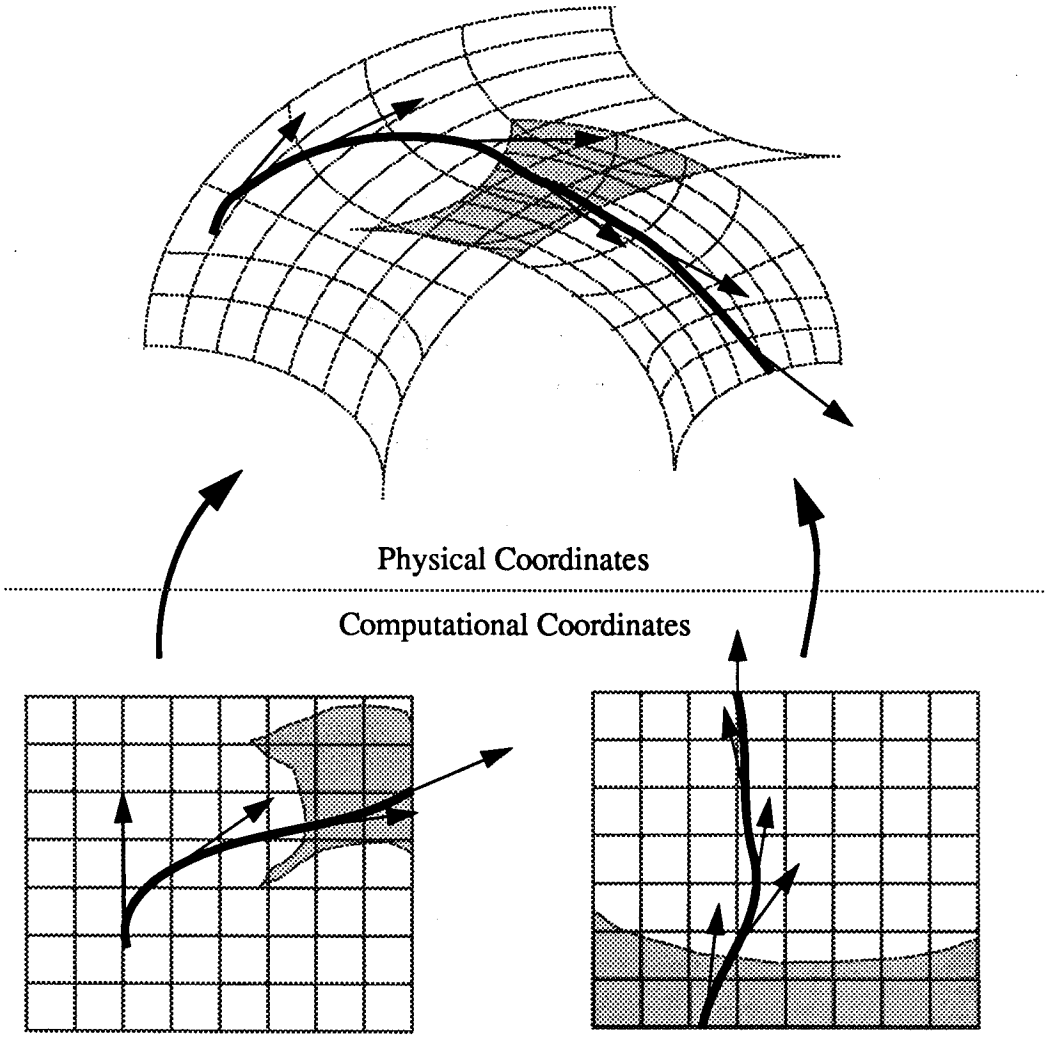

Fig. 4: Streamline computation typically requires integration of a vector field across grid boundaries.

where $x \in B$, $v \in V$ and g is a general linear transformation. In terms of our carpet-gluing analogy, we glue backing to backing and fiber to fiber, but in the process we can stretch, rotate, or twist the fibers. This added twist in the vector spaces is the essential topological feature of vector bundles.

Although not necessary, it is usually convenient to use the coordinate-chart neighborhoods of the manifold to define the charts for the bundle. In particular, it will be convenient for the class library we develop in the next section, so we assume this for the remainder of the article. We can then compose the manifold charts with the bundle charts, and think of the transition functions as mapping the coordinates of the base point and the components of the vector in one chart to the coordinates and components in another chart, twist included.

A section of a vector bundle is formally a smooth map from the base space B to the total space E. It picks a value in the vector space attached to each point of the base space. In terms of the charts, a section determines specific component values in each chart.

Sections support vector algebra. We can add two sections of the same bundle by adding the vectors at each point. Similarly, if we have a smooth real-valued function defined on the base space, we can then multiply a section by the function by multiplying the vector value of the section by the scalar value of the function at each point. Multiplication of a section by a scalar function is very similar to multiplication of a simple vector by a simple scalar, and vector algebra thus essentially propagates from the vector space to the bundle.

In addition to vector algebra, sections support a number of other operations that are useful in computation and visualization. They support calculus; fields can be differentiated and integrated. They support a variety of composition and decomposition operations; fields can be sliced or glued together. Vector bundles are closed under these operations. Both the operands and the results are sections of vector bundles. Unfortunately, the description of these operations requires more mathematical structure than we have room to describe in this article.

The section of a vector bundle is the top level in the natural hierarchy of structure in the vector-bundle data model. We will now translate this mathematical hierarchy into an object-oriented class hierarchy.

Vector-Bundle Class Hierarchy

The mathematical hierarchy described in the preceding section can be considered a prescription for a library of abstract classes. In this article, we cannot give a detailed specification for all the classes. Instead, we will describe simplified specifications for the central classes in enough detail to support the later discussion of specific representations and visualization functions. We will start with the most general class—point set—and proceed to the more specific. Our specifications will be written using the Eiffel programming language.[2]

Point set. As described above, a point set is a set of points, and points are named, but otherwise featureless, objects. We will assume that this set is countable so that we can index the set—in other words, name the points with integers. This assumption holds for a wide range of applications, and it simplifies the presentation. More general assumptions require more complex specifications and exceed the space available in this article.

An indexed set is very convenient for programming. It allows us to use the name of the point as an index into arrays, lists, or other data structures. This scheme is quite general: points are identified with indices or references, while functions of points, for instance coordinates, are identified with the value at the location indicated by the index or reference. This scheme makes class PT_SET quite simple:

```
deferred class PT_SET
    -- abstract indexed set of points
feature
    pt_ct: INTEGER;
    -- point count, the number of points in the set
    pt_itr: PT_ITR is deferred end;
    -- returns an iterator for the set.
end; -- class PT_SET
```

Class PT_SET is abstract, "deferred" in Eiffel, because we have not committed to any mechanism for storing or otherwise representing the members of the set. Nor have we specified a complete set of operations for PT_SET—just those we need later. In particular, we have not specified any mechanism for inserting or deleting members, or for forming subsets.

The class PT_ITR is an abstract iterator for the PT_SET. Iterators are a generalization of the FOR loop available in most procedural languages. Once initialized, an iterator will return an element of a collection each time it is called, until all items of the collection have been returned. Iterators hide the representation of a collection, its internal structure and indexing methods, while allowing a client to access each part. Most object-oriented languages do not provide a specific language mechanism for iterators; they are constructed as ordinary classes. To iterate over our collection of points we have:

```
deferred class PT_ITR
    -- abstract iterator over point sets
feature
    domain: PT_SET;
    -- the point set to iterate over
    point: INTEGER;
    -- the current point
    next is deferred end;
    -- sets point to the next member of the domain
    -- also sets done if iteration complete
    done: BOOLEAN;
    -- true if iteration complete
    reset is deferred end;
    -- resets iteration to first member of domain
end; -- class PT_ITR
```

The operations *next* and *reset* are deferred because we cannot implement them without knowing the representation of PT_SET.

Together, PT_SET and PT_ITR provide a simple abstraction: point sets we can iterate over.

Topological space. A topological space is a point set with a notion of neighborhoods, or connectivity. Our topological space class TSPACE inherits PT_SET and adds some neighborhood features. We proceed in the same

spirit as PT_SET; we specify an abstract class TSPACE that provides the essential operations, without committing to any representation. A suitable specification is:

```
deferred class TSPACE
    -- abstract topological space
inherit
    PT_SET
feature
    nbrhd_ct: INTEGER;
    -- neighborhood count, the number of neighborhoods
    -- specified
    nbrhd_itr: NBRHD_ITR is deferred end;
    -- returns a iterator for the neighborhoods in the topology.
end; -- class TSPACE
```

The class NBRHD_ITR is an abstract iterator for the neighborhoods of TSPACE. It has a specification very similar to PT_ITR, except that feature *domain* is of type TSPACE, and feature *point* is replaced by a feature *nbrhd* which returns an array of integers, each a point in the neighborhood.

TSPACE and NBRHD_ITR extend the abstraction defined by PT_SET and PT_ITR to provide neighborhoods we can iterate over.

Manifold. A manifold is a topological space with coordinate patches and coordinate transformations. Class MANIFOLD inherits TSPACE and adds coordinate features:

```
deferred class MANIFOLD
    -- abstract d dimensional manifold
inherit
    TSPACE
feature
    d: INTEGER;
    -- the dimension of the manifold
    chart_ct: INTEGER;
    -- chart count, the number of charts specified
    chart(c: INTEGER): MCHART is deferred end;
    -- returns the cth chart
    xfcn(c1, c2: INTEGER): MXFCN is deferred end;
    -- returns transition function from chart c1 to chart c2
end; -- class MANIFOLD
```

Again, operations *chart* and *xfcn* are deferred because we have not committed ourselves as to how the charts and transition functions of the manifold are represented. We have, however, specified how the information in individual charts and transition functions is accessed, namely through the classes MCHART and MXFCN, respectively:

```
deferred class MCHART
    -- manifold coordinate patch
feature
    coord(p:INTEGER): ARRAY[REAL] is deferred end;
    -- coordinates of point p
    coordinv(r: ARRAY[REAL]): INTEGER is deferred end;
    -- returns point with coordinates r
    in_range(r: ARRAY[REAL]): BOOLEAN is deferred end;
    -- true if coordinates r in range of chart,
    -- sets adjoining if false
    adjoining: INTEGER;
    -- index of adjoining chart containing r if in_range false
    -- returns 0 if no adjoining chart
end; -- class MCHART
```

```
deferred class MXFCN
    -- manifold transition function (coordinate transformation)
feature
    domain, range: MCHART;
    -- domain and range of coordinate transformation
    value(r: ARRAY[REAL]): ARRAY[REAL] is deferred;
    -- returns coordinate in range corresponding to coordinate r
    -- in domain
end; -- class MXFCN;
```

The MCHART feature *adjoining* provides an index for an adjoining chart, similar to the annotations usually provided in the margins of each map in a geographic atlas. The various features of MCHART and MXFCN are deferred because we have made no decision about how the coordinates are represented internally—only how they are accessed.

Classes MANIFOLD, MCHART and MXFCN extend the abstractions defined by TSPACE and NBRHD_ITR to provide coordinate patches and transformations between patches.

Vector bundle. A vector bundle has a total space, a base space, a projection function, coordinate patches, and transition functions. We will identify the class VBDL with the total space and, by analogy with common mathematical practice, we will consider the projection function and other structures to be features of this class. The class VBDL differs from the mathematical structure in one important respect: we do not represent the points of the total space. Representing all points of the total space would not serve any useful purpose—we are only interested in representing the points in a cross section of the total space. Class VBDL thus becomes a repository for information we want to share among several instances of class VSEC (see below), and a factory for allocating the charts used by VSEC to store and access the points in a cross section.

```
deferred class VBDL
    -- abstract vector bundle
feature
    db, df, dt, d: INTEGER;
    -- the base, fiber, total and bundle dimension
    base: MANIFOLD
    -- the base space
    chart_ct: INTEGER;
    -- chart count, the number of charts specified
    chart(c: INTEGER): VCHART is deferred end;
    -- allocates and returns a copy of the cth chart
    xfcn(c1, c2: INTEGER): VXFCN is deferred end;
    -- returns transition function from chart c1 to chart c2
end; -- class VBDL
```

Classes VCHART and VXFCN have specifications similar to MCHART and MXFCN, respectively, but have different implementations, according to the mathematical structure discussed above. The assumption made previously, that the chart neighborhoods were the same for the base space and the bundle, makes it convenient to have VCHART return only the vector-space components, and similarly for VXFCN. The base-space coordinates can be obtained from the manifold charts. Both VCHART and VXFCN are deferred because no representation has been chosen.

Section of a vector bundle. Class VSEC is the data type of central interest. A section of a vector bundle is to-

pologically equivalent to the base space, which means that as a topological space, and hence as a point set, any section is identical to the base space. In particular, this means that the point and neighborhood iterators for the base space can be used for any section. As a result, class VSEC is very similar to class MANIFOLD, except that the charts return components for the vector space, that is, the dependent variables of the field.

```
deferred class VSEC
    -- abstract section of a vector bundle
feature
    domain: MANIFOLD;
    -- the base space, obtained from the bundle
    range: VBDL;
    -- the bundle to which the section belongs
    chart(c: INTEGER): VCHART is deferred end;
    -- returns the cth chart
    scaleby(a: SFCN): VSEC;
    -- multiplies the section by the scalar function a;
    add(other: VSEC): VSEC
    -- adds the current section to other and returns the result.
end; -- class VSEC
```

When it is created, VSEC must define the values of the dependent variables, but this is a representation-dependent issue, so an operation *chart* is deferred. The class SFCN models scalar functions defined on the same base space, but we will not need to specify it further.

As specified, class VSEC allows us to access the data in a section of a vector bundle, and to scale a section or add two sections. Many other operations could be specified, most notably the operations of calculus, but space limitations preclude a more complete specification in this article.

Representation

The classes described above are abstract. A number of design decisions must be made to produce representations of these classes. The various data structures commonly used in scientific programming correspond to different choices for these decisions.

There are three primary representation issues: (i) the dimension of the base and fiber spaces; (ii) the representation of connectivity information in class TSPACE; and (iii) the representation for class CHART and the chart operations in classes VBDL and VSEC. As an example, we will describe how these issues are resolved in typical finite-difference applications.

In the finite-difference method, the base space is usually represented by one or more regular grids called blocks. We will discuss the single-block case first, then indicate the extension to multiple blocks.

The grid is regular in the sense that each interior node has the same number of nearest neighbors, namely $2d$, where d is the dimension of the base space. This grid can be stored in an array with d indices.

It is well known that multiple-index arrays can also be indexed with a single index that just gives the offset from the base of the array. Each point in the grid is thus uniquely associated with an offset and the PT_ITR class can be implemented using this fact; it just increments through all the offsets in the array.

The *d* indices, or floating-point conversions of them, usually define the "computational" coordinates for the base space. The computational coordinate system is a coordinate chart for the base space, and the MCHART class calculates the coordinate indices from the point offset.

The "physical" coordinates map the points of the grid into the real, physical space of the application. The physical coordinates are not necessarily coordinate charts as defined for manifolds above. For instance, the computational space may be two-dimensional and the physical space may be three-dimensional. For this reason, we prefer to think of the physical coordinates as a vector field on the manifold, and represent this field using class VSEC.

The *d* index array representation has the further desirable feature that neighboring points can be found by calculation: if a point has indices I, J, K..., then its neighbors are at $I \pm 1$, $J \pm 1$, $K \pm 1$..., etc. Stated in our manifold language, the neighborhoods can be calculated from the charts. The NBRHD_ITR class can thus be implemented in terms of the charts.

Finally, the VCHART class stores the section values in multiple index arrays, indexed by the point offset or the manifold coordinates.

Multiblock finite difference is essentially the same as single block, except there are multiple grids, each with its own computational coordinates and hence multiple coordinate charts. As described previously, coordinate transformations between the computational coordinates are maintained, and class MXFCN can use them. Class VXFCN is implemented using class MXFCN and the numerical Jacobian of the coordinate transformations. In some cases, explicit interblock connectivity information may be stored as well, and NBRHD_ITR can use this information.

Visualization

We return to the streamline example discussed earlier, and show how a representation-independent visualization algorithm can be constructed using the interface defined by the abstract classes. The computation of the streamline by integration of the vector field can be accomplished by the following algorithm:

```
streamline(vfld, pfld:VSEC, p0, max_ct: INTEGER) is
    -- computes streamline of vfld starting at p0
    -- and maps it into the physical coordinates
    -- given by pfld defined on same base;
    -- stops after max_ct points have been drawn
    local
        r1, r2, v1, v2, x1, x2: ARRAY[REAL];
        p1, p2, c, cadj, ct: INTEGER;
        base: MANIFOLD;
    do
        base := vfld.domain;
        p1 := p0;
        from
            ct := 1;
            c := 1;
            -- assume p0 in first chart for simplicity
            r1 := base.chart(c).coord(p1);
            -- get coordinates of point
            v1 := vfld.chart(c).coord(p1);
            -- get value of field
            r2 := r1 + integrate(v1);
            -- find coordinates of the next
            -- point on the streamline
            -- using one of the calculus operations
            -- we haven't specified
        until
            c = 0 or ct > max_ct
        loop
            if base.chart(c).in_range(r2) then
                -- r2 was in range of current chart
                p2 := base.chart(c).coordinv(r2);
                -- find point with coordinates r2
                x1 := pfld.chart(c).coord(p1);
                x2 := pfld.chart(c).coord(p2);
                draw_line(x1, x2);
                -- draw streamline in physical space
                ct := ct + 1;
                p1 := p2;
                -- move to leading point
                r1 := base.chart(c).coord(p1);
                v1 := vfld.chart(c).coord(p1);
                r2 := r1 + integrate(v1);
                -- find the next point on the streamline
            else
                -- r2 lies outside range of current chart
                -- move to the adjoining chart and try again
                cadj := base.chart(c).adjoining;
                v1 := vfld. xfcn(c, cadj).value(r1, v1);
                r1 := base. xfcn(c, cadj).value(r1);
                r2 := r1 + integrate(v1);
                c := cadj;
            end;
        end;
    end;
```

The algorithm starts at the given initial point, and integrates along the streamline until it runs off the current chart. It then invokes a transition function to the adjoining chart, and continues. The process is repeated until there is no adjoining chart, or the requested number of points have been drawn. This algorithm is entirely independent of the representation chosen for the various classes, and it is optimal in the sense that it makes full use of the geometric structure of the data.

Capturing Abstractions

We have described the mathematical structure of vector bundles, and shown how they can be used to define a collection of abstract classes useful in scientific visualization. The particular class specifications we have given are at best a sketch of the full specifications required for practical implementation, but they do demonstrate that the object-oriented paradigm provides a number of mechanisms for capturing scientific abstractions directly in code. It is our opinion that the object-oriented approach will substantially increase the level of abstraction in future scientific codes.

Acknowledgment

This work was supported in part by the United States Department of Energy under contract no. DE-ACO4-76DP00789. ■

References

1. D. M. Butler and M. H. Pendley, *Comp. Phys.* 3 (5), 45 (1989).
2. B. Meyer, *Eiffel: The Language* (Prentice-Hall, NY, 1991).