Interactive Quantitative Visualization

by R. L. Peskin
S. S. Walther
A. M. Froncioni
T. I. Boubez

Interactive Quantitative Visualization, a methodology to enhance scientific and engineering computational simulation prototyping, is defined. Appropriate strategies for implementing IQV in a workstation-based distributed computing environment are discussed. Object-oriented graphical tools and a new data management technique to support IQV and computational steering are described. Two examples of IQV and computational steering are presented: 1) a system to allow interactive solution and visualization of nonlinear boundary-value problems; and 2) a modeling exercise illustrating how IQV and computational steering are used together to prototype simulation of a complex physical system, namely a flag flapping in the wind.

Introduction

Our objective in this research program is to develop strategies for Interactive Quantitative Visualization (IQV). We mean by this the ability to dynamically present results of scientific and engineering computations in a graphical (e.g., visual) format and also to allow the user to extract accurate quantitative information directly from the visual presentation. One important need for such a capability arises in simulation model development and prototyping; this activity requires continuous interaction between the computational model and visualizations of its data output. For example, an aerodynamics engineer may need to simulate turbulent flow over a wing. His computational model consists of a set of partial differential equations and a flow-region (geometry) specification. However, in order to describe the turbulence, he uses approximations in his basic equations, and the prototyping process involves computational assessment of the validity of these approximations. To do this, the engineer initiates computations, and, after an appropriate number of computational steps, examines a visual (graphical) representation of the interim results. If he notices some questionable features, he uses his interactive pointing device to retrieve the actual numerical data associated with the locality of these features. After examination of this retrieved dataset, and perhaps some subsidiary computation using this set, he may revise his approximations, modify his model equations accordingly, and proceed with the simulation using the newly revised model. We refer to this sort of procedure, which requires incremental data visualization and model modification, as *computational steering* [1]. It should be noted that we are concerned with steering the model itself, not the specific numerical methods and procedures employed in effecting a solution.

IQV is also needed to supplement information obtained from visual feature recognition when more

traditional postprocessing visualization is used in the supercomputing environment. New hardware capabilities in computer graphics, coupled with the ability of supercomputers to produce massive amounts of data, have created both the need and the capability to use graphics and visualization in the analysis of data. Dramatic visual graphic effects (both static and, in a few cases, dynamic) have been produced. In some instances these have been useful in identifying physical phenomena. Often, however, the exotic graphics obscure the underlying quantitative information. The extraction of quantitative information from a complex rendered volumetric scene is a difficult problem. IQV strategies are directed toward data structure manipulation and pictorial representation that allow access to quantitative information necessary for the user to judge the viability of his simulation. Interaction with the visual output, use of color, rotation, translation, scaling, and other features are all designed to enable the user rapidly to obtain quantitative information that can be used for model evaluation in the case of postprocessing, or for *in situ* changes in the prototyping case. At present, IQV and some of the more complex visualization techniques should be viewed as separate but complementary methodologies; as visualization technology advances, we expect that these methodologies will merge.

There has been much attention paid recently to the area of scientific visualization; the field was brought into focus by a National Science Foundation report [1]. This report, which stressed the need to bring visualization techniques to bear on the problem of large supercomputer dataset analysis, also emphasized the necessity of developing computational steering methodologies to improve prototyping and effect better interaction between the user and his numerical simulation. Our research is aimed at developing a computational steering environment for computational fluid dynamics (CFD). We feel that meaningful experimentation in quantitative visualization and computational steering requires domain specificity; it is unlikely that applicable techniques can be developed solely in the abstract. However, we do broaden the scope of CFD to include not only traditional Navier–Stokes equation simulation, but problems in nonlinear dynamics and differential equations which are applicable to the CFD and related fields. Consequently, we expect our work to be adaptable to other technical domains.

In the following sections, we first consider interactive, quantitative graphics from a historical perspective. We then describe the overall system environment which is being developed to support our prototyping framework, and we discuss some of the quantitative visualization tools which have been developed. These are useful for both prototyping environments and postprocessing. Next we discuss the general data management strategies we have developed, and how they are implemented in our object-oriented environment. As examples of IQV and computational steering, we describe in some detail a differential equation solution system and the development of a physical model (a flag flapping in the wind) in the context of how that development employs IQV. We also show examples of a more complex CFD tool set and some flow-field applications. Finally we draw some conclusions about our concepts and their future directions.

Previous related work

Although scientific computer graphics was originally implemented as subroutine calls to be included in the user's program, and, in fact, this mode remains the predominant one today, interactive packages appeared in the late 1970s. Among the first were the EZGraphing[1] package from Tektronix Corporation and its successor, PLOT10IG[1] [2]. These were written in FORTRAN, and were meant to be used with TEKTRONIX[1] display terminals. The UNIX[2] Plot libraries are similar in functionality to these PLOT10[1] programs. The UNIX libraries allow easier interfacing with computational programs through the use of pipes, etc., but the user interface to the graphics is command-line based. Furthermore, the UNIX Plot libraries are limited in variety of graphical representation. With the advent of the X-Windows System,[3] there has been improvement in both the user interface and the spectrum of graphic tools available. Designed primarily for interactive control of postprocessing display, these products introduced concepts, such as dynamic scaling, that remain part of a modern interactive display program. However, user interfaces for the early systems are difficult to learn and use. In addition, they do not provide for dynamic interaction between the computational component and the graphics. Such dynamic interaction is necessary if the user is to be able to interactively modify the numerical computation while simultaneously being able to view the results of the modification. Nevertheless, these traditional interactive packages established the desirability of being able to manipulate visualizations of computed data without being constrained to pre-bound subroutines.

The advent of the personal computer resulted in the generation of numerous interactive graphics products, so many that it is futile to discuss them in detail here. In general, the "number-crunching" limitations of the PCs implied that the interactive graphics systems were designed to emphasize manipulation of already

[1] EZGraphing, PLOT10IG, TEKTRONIX, and PLOT10 are registered trademarks of Tektronix, Inc.

[2] UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

[3] X-Windows is a trademark of MIT.

computed or measured data. In some cases these datasets were in file format, and in a few cases the data were available from "on-line" instrumentation interfaces. As in other areas of software, many of these PC graphics systems serve as paradigms for systems that would be desirable on workstations. They contain many features which promote interactivity and experimentation with the data they visualize. However, many of these PC systems are written in machine language and thus do not readily conform to user needs. For example, three-dimensional surface representation might be part of such a PC package, but the user would not be able to rotate the surface. With the advent of increased computation power in the PCs, more user options are appearing. Recent spreadsheet products now integrate graphics, and, in so doing, allow the user some access to interactive quantitative visualization. For example, a new product, Wingz[4] [3], not only promotes interactive visualization of data, but provides features such as three-dimensional surface rotation and transformation normally found on more complex hardware–software systems.

There have been some recent efforts to provide more interactivity in systems primarily used for postprocessing visualization. DAVID [4] runs under X-Windows, and was designed for two-dimensional computational fluid dynamics data visualization. DAVID supports such features as sequential views of time frames ("movie views"), and provides tools used to view data in a variety of ways, e.g., cross-sectional graphs, contour plots, and surface plots. Similar capabilities are available with the NCSA Software Suite [5]. More powerful graphics can be obtained by combining X-Windows and direct hardware graphics support. By use of X-Windows controllers and screen regions devoted to display of native graphics, the Ardent[5] computer is able to provide a degree of animation speed response to user manipulation of dial boxes [6]. Some direct coupling of scientific computation and visualization is possible with "mini-supercomputers" such as the Ardent or Stellar[5] products. User feedback to adjust computational parameters is possible, but flexibility is limited by the need to accommodate the feedback requirements at compilation time.

Some recent work in enhancing simulations of physics phenomena with visualization has included some limited interactive capabilities. The paper by Rapaport [7] shows an example. Additionally, there exist interactive computer tutoring systems that rely on interactive graphics which are strongly coupled to physical models. An example is found in the paper by Bourne et al. [8]; this paper is of particular interest because it describes an implementation of the tutoring system in Smalltalk-80.[6]

The IQV system to be discussed in this paper traces some of its origins to work by Walther [9] in the development of quantitative graphics tools for UNIX workstations. These tools provided a uniform user interface to a set of linear graphing programs and three-dimensional graphing programs. Based on a common set of algorithms, these tools have now been enhanced and implemented in Smalltalk-80. The primary characteristic of this latest implementation is its use of the Smalltalk *Model-View-Controller* paradigm as the basic mechanism supporting a wide latitude of user options for quantitatively based data visualization. Furthermore, customization is straightforward and does not require alteration of the underlying structures holding the user's data. In effect, the same interface and graphic tool set is available for either file-based data (postprocessing) or data computed *in situ*. These tools, then, are capable of incorporation into a computational steering environment. There seems to be little in the literature reporting actual computational steering systems. A prototype computational steering example was developed by Campbell for two-point boundary-value problems;[7] it was implemented under Suntools.[8] We later discuss a similar differential equation system we have developed using IQV techniques and having some computational steering capabilities. Our overall research direction *vis-à-vis* IQV and its relation to computational steering is described in the paper by Peskin, Walther, et al. [10].

System overview

The computational system that is being developed to support the IQV research is based on a distributed computing concept: a graphics workstation in communication with one or more "back-end" high-power computers. A discussion of the technical aspects of the system can be found in the paper by Walther and Peskin [11]. As described in [10] above, a prototyping environment that is dedicated to scientific simulations must combine ease of use with flexibility. At the same time, performance must be at a sufficient level to support true interactive computing. The well-known FPI (features, performance, interface) of Macintosh[9] programming [12] apply in the scientific simulation environment.

The workstations employed are common bit-mapped UNIX systems. We are using either the Sun

[4] Wingz is a trademark of Informix Software, Inc.

[5] Ardent, Titan, and Stellar are trademarks of Stardent Computer, Inc.

[6] Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

[7] J. A. Campbell, Los Alamos Scientific Laboratories, Los Alamos, NM, 1989, private communication.

[8] SunTools, Sun Microsystems, and SunView are trademarks of Sun Microsystems, Inc.

[9] Macintosh is a trademark of Apple Computer, Inc.

Microsystems[8] stations or the Tektronix 4317-19 color stations. These stations typically have 12 megabytes of memory and large (19-inch) screens. (In addition, we have implemented some of the IQV capabilities on the Ardent Titan[5] workstation. This machine has vector hardware and advanced color graphics capabilities.) Key to our system implementation is the use of Smalltalk-80 [13] as the user interface on the workstation. Depending on the platform, we are able to take advantage of color graphics under Smalltalk, as well as all of its other features. Of greatest importance to us is the utilization of the *Model-View-Controller* paradigm. This allows a tight coupling between model (simulation representation) changes, the graphical view of those changes, and user interface via keyboard and/or pointing device. See Peskin, Walther, et al. [10] for a more detailed discussion of the use of the MVC in scientific prototyping. This reference also discusses the merits and drawbacks of Smalltalk as a scientific computing environment. For the purposes of this paper, it is important to note that Smalltalk provides a highly productive environment under which we are able to experiment with new concepts and strategies for IQV.

Smalltalk's major liability in scientific computation is its performance; the true object-oriented features of the language together with its incremental compilation and dynamic binding produce computational performance about an order of magnitude less than statically compiled C code. This results in very severe restrictions in those scientific models that require multiple iterations. Smalltalk itself has high performance in its graphics and character scanner operations; this is accomplished by use of primitives, i.e., machine code. While there is significant work being done in improving Smalltalk performance [14, 15], our approach has been to use primitives to perform numerically intensive computations while still in the Smalltalk environment. In some cases, these primitives perform machine-code operations on the workstation itself. However, as is appropriate to a scientific computing environment, our primary use of primitives is to invoke numerically intensive computations on a "back-end" computer via distributed computing which employs a networking protocol.

Our distributed environment consists of a collection of parallel and serial high-performance computers, specifically a 128-processor NCUBE[10] hypercube, a 32-processor Butterfly[11] GP1000, and a two-processor Ardent Titan. The networking protocol allows the user's interface system to request appropriate computation services from one or more of these machines. In the future, "intelligence" in the Smalltalk system will choose resources on the basis of most appropriate functionality. At present we are concentrating on establishing reliable distributed networking; we use the Butterfly for specialized parallel back-end processing and the Ardent for more routine numerical processing. In order to effect transparency, we are currently installing C-Linda [16] on the Butterfly and NCUBE, and plan to develop methods to allow Smalltalk to initiate Linda programs over the distributed system. In a technical sense, these back-end processors perform "methods" on behalf of Smalltalk objects. **Figure 1** presents a diagram of the current configuration of our distributed computational environment.

• *Graphical visualization tools*

As indicated above, the interactive visualization tools we have been developing are described in detail in several other forums [9–11]. The earliest versions of our user interface [9] serviced monochrome and color graphics terminals through libraries of device-independent 2D and 3D graphics functions, written in C, which provided both interactive and subroutine-callable graphic tools for scientific and engineering data analysis. These libraries were extended to support bit-mapped workstations under SunView[8] and X-Windows and to make use of certain experimental environments using multiple processors [17].

In these iterations, the tools were organized *functionally* around a set of visualization behaviors; the 2D graphing methods presented data according to the Graphics Kernel Standard (GKS); the 3D graphing methods used standard transformation and perspective techniques to present data as three-dimensional wire-frame surfaces. But while these functions were *modular*, the structure was not yet *object-oriented*; thus, although coded functions were reusable and software could be extended by adding routines to existing libraries, the logical relations between graphical behaviors did not support inheritance and encapsulation. Most significantly, the event-management facilities of the interfacing environments (e.g., Suntools or X-Windows), lacked a good paradigm for implementing *intelligent* process control, that is, interaction engineered jointly by the user viewing the data *and* by the data itself.

Under Smalltalk-80, the *Model-View-Controller* paradigm provides a process-control facility through which data objects (models) can deploy a variety of visual presentation behaviors (views) and can be responsive to interruptions from a user (controllers). The prototyping of the basic scientific visualization repertoire in Smalltalk is described in [10] and in [11].

Figure 2 shows the present organization of visualization classes. DataModel is the class that

[10] NCUBE is a trademark of NCUBE Corporation.

[11] Butterfly is a trademark of Bolt, Beranek and Newman, Inc.

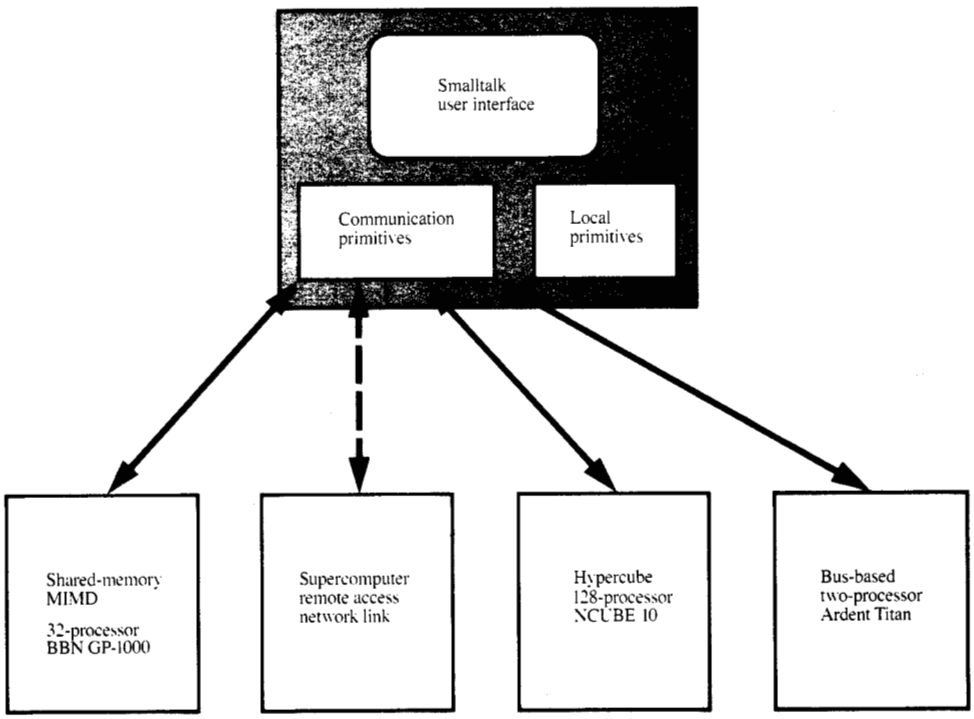

Figure 1

Current configuration of distributed computational environment.

comprises basic 2D graphing methods (Graphics Kernel Standard compatible). DataView contains the display methods supporting these 2D presentation methods. DataSurface is the class that comprises basic 3D graphing methods, inheriting from DataModel such graphing methods as are common to them both. DataSurfaceView contains the display methods supporting 3D presentation methods. DataContour comprises methods for the presentation of 3D data as two-dimensional cross sections. DataVector uses the 2D graphing methods of DataModel, adding some of its own. StreamData uses 3D graphing methods, again with its own additions. The view supporting DataVector's display needs, VectorView, is a subclass of DataView (the basic two-dimensional view). DataSurfaceView comprises display behaviors supporting three-dimensional representation. Each view class has a controller to manage requests for view behavior and/or model behavior.

Visualization classes (Model–View–Controller paradigm)

VisualizationObjects (*Models*)	**DisplayBehaviors** (*Views*)	**Managers** (*Controllers*)
Object	View	MouseMenuController
DataModel	DataView	DataViewController
DataSurface	DataSurfaceView	DataSurfaceViewController
PrimitiveSurface		
Flag	FlagView	FlagViewController
StreamData	StreamView	StreamViewController
DataContour	ContourView	ContourViewController
DataVector	VectorView	VectorViewController
LogisticEquation		
Oscillator		
TwoPointBVP	TwoPointBVPView	TwoPointBVPController
XYData		

Figure 2

Graphical interface classes for scientific modeling. Subclassing is represented through indentation of class names.

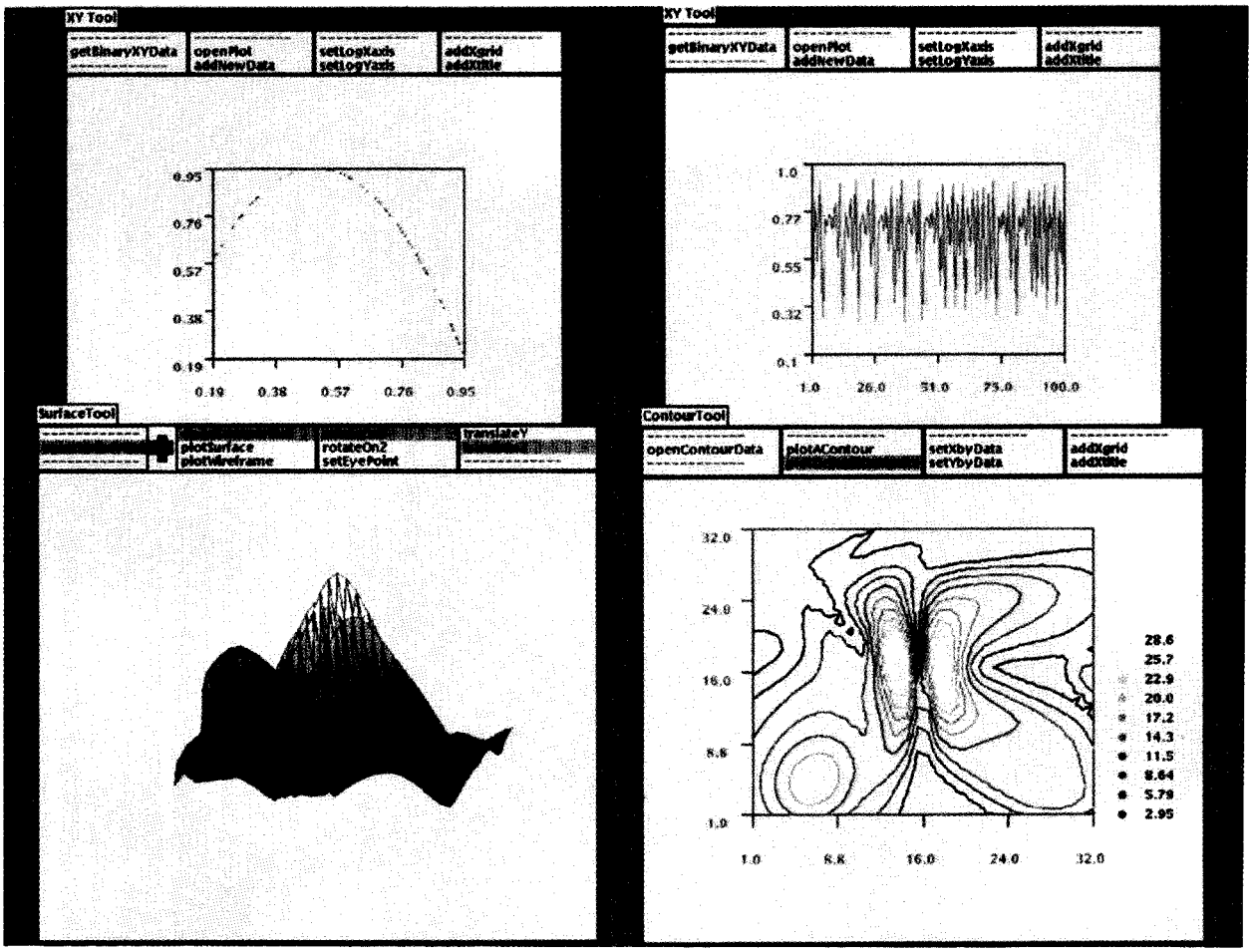

Figure 3

Sample visualization utilities. Clockwise from top left: LogisticEquation running a phase plot, XYTool, ContourTool, SurfaceTool.

Any object that needs some numbers graphed can declare an instance of DataModel, DataSurface, or any other visualization class, and can thereby make use of that class's graphing methods directly as it is computing. In effect, computational objects can *deploy* visualization objects in the same way that any object in Smalltalk can deploy other objects for its own purposes. This is one of the most powerful features of true object-oriented environments; the total repertoire of described objects is accessible at all times and is ready to perform on behalf of any request properly made. The impact of this structure for scientific visualization is that computational elements are neither isolated from the visualizing apparatus nor forced to become specialized to only one representation format. Moreover, as visualization classes are added, any computational object that has (or can generate) the appropriate information can immediately avail itself of that capability.

In addition, each of the views described above provides a class method to open an instance of itself as a stand-alone tool that can access and display precomputed data files. A "sampler" of visualization utilities is presented in **Figure 3**. **Figure 4** shows *Vectortool*, a view opened on an instance of DataVector, the class that knows how to represent two-dimensional vector data (position and velocities). In that figure, the view has been invoked as a stand-alone *postprocessing* visualization tool to explore data resulting from a parallelized computation of a two-dimensional flow.

Another handy technique to effect computational steering is to subclass a computational model directly off one of these visualization classes so that the graphic behavior can be inherited or modified. Normally, this would mean that such an object has computation methods of its own to generate graphable data and that it will add initialization and control methods suitable to its

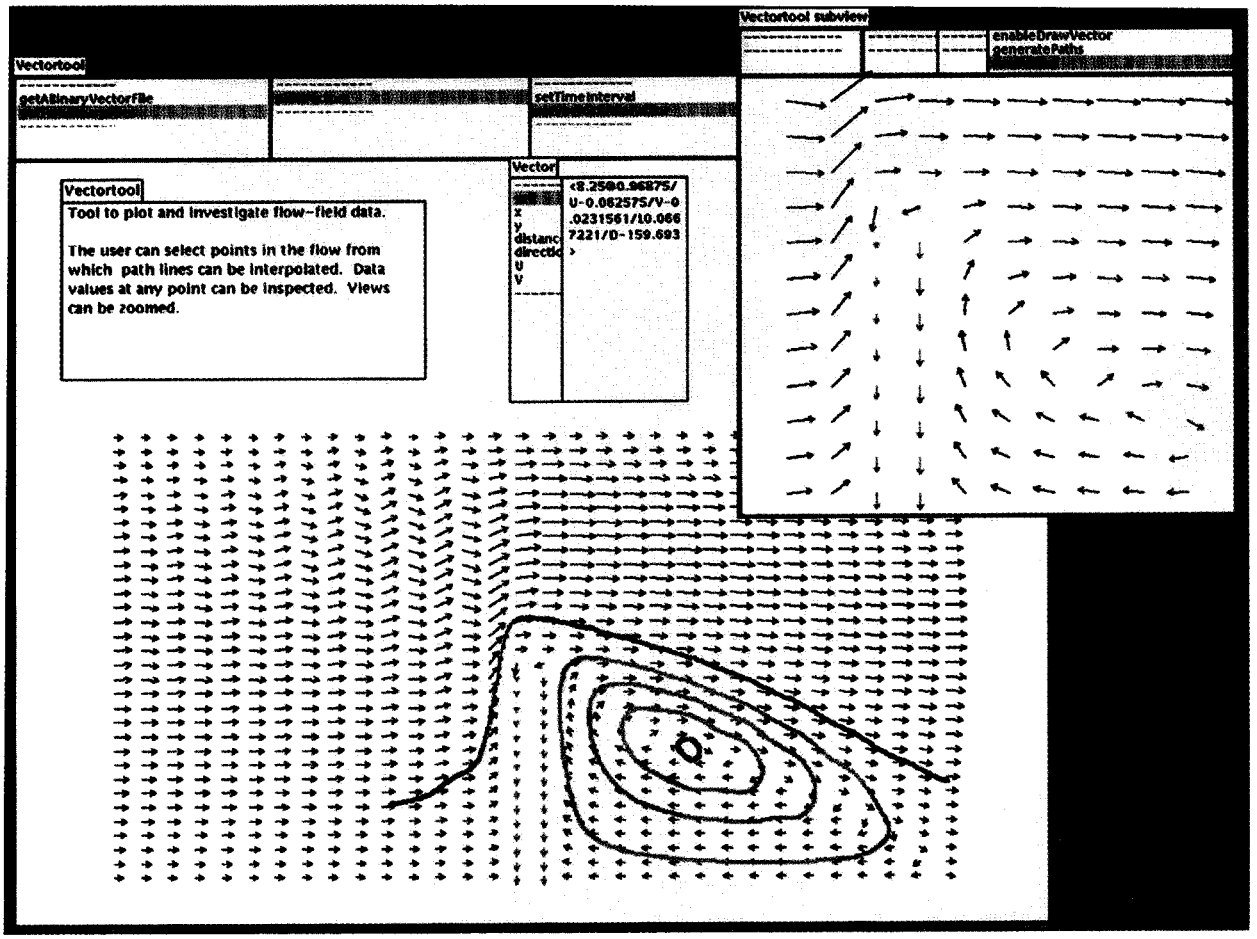

Figure 4

VectorTool. Features shown illustrate vector flow, zoom, path interpolation, and retrieval of data values. The inspection box shows the position and velocity values of the vector by the red dot in the zoomed view.

user interface requirements. As shown in Figure 2, the classes LogisticEquation, Oscillator, TwoPointBVP (described in the sections on the ODE solution tool), and Flag (described in the last example section) are computational objects taking the subclassing route to their own visualization.

A key feature of this visualization environment is the *retrievability* of original data values from any graphic effect. Put another way, a visual effect is of value primarily because it is informative and/or can be manipulated to yield further visual effects that, for example, lead the user to derivable quantities or to identification of structural features. **Figure 5** illustrates the multiple zoom capability of XYtool which allows the user to explore regions in a visualization based on a computation of the Henon attractor [18]. Zoomed regions can be zoomed further themselves, and so on until the resolution of the data is exhausted. The user is completely free to probe any region of any view or subview as many times as desired; all the original data are retained, and only original computed values are used. In Vectortool, shown in Figure 4, the user can zoom on any area of the flow field, retrieve original data values of any vectors, or generate path lines from any point in the flow.

Moreover, the view paradigm is itself quite adaptable for visualizing multiple objects acting concurrently and for tracking their behavior. We have been experimenting with a "spreadsheet" graphical interface, based on the MVC paradigm in which a master view of integrated subviews is opened on a "matrix" of models. Computational objects (the models in the cells) have access to the full repertoire of graphic visualization described above and can also display icons or flash text

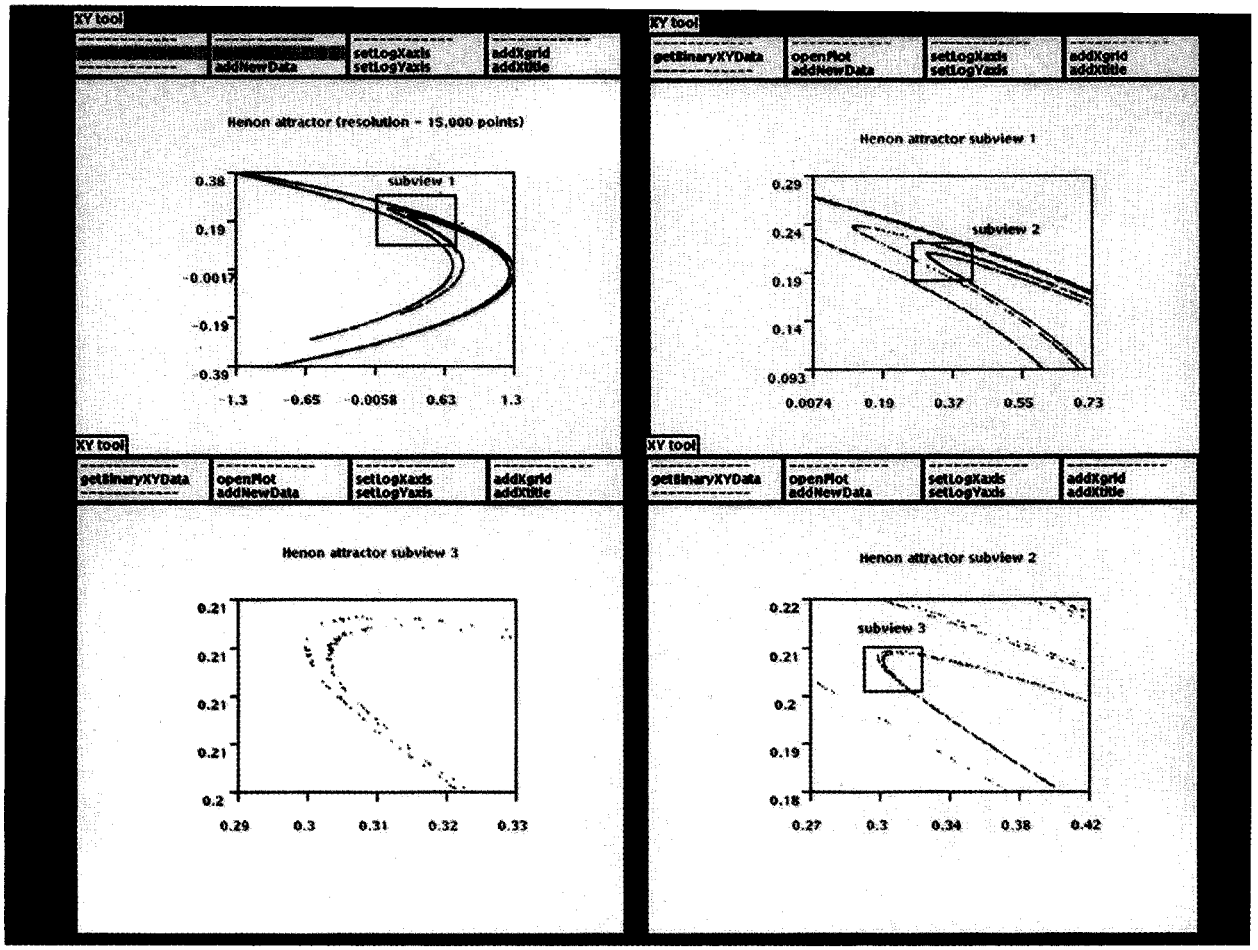

Figure 5

XYTool graphing Henon attractor data. XYTool has been used to zoom in on regions of the data that are of interest. Each zoomed view can itself be zoomed. Clockwise from top left, the views explore orbital structures whose substructures are themselves orbital.

messages to report specific values or to indicate their status. In addition, the user can open another view, with a control panel on a cell object, so as to manipulate the view interactively. **Figure 6** shows a spreadsheet on a 2×2 matrix object (a domain decomposition problem) in which each cell object (the subdomains) has deployed surface-plotting display methods. As the subdomain computations progress, they update the views which represent their solution at the time for their region of the problem space. The processes that are computing the regions can be executing on different processors from the processor which is running the graphics. The individual views are updated in response to messages sent by the cell objects from wherever they are computing. Currently, the messaging is implemented through remote procedure calls using sockets as provided for in 4.3BSD[12] UNIX [19].

• *Data management techniques for IQV*

If incremental quantitative visualization is to accomplish the graphical representation of a computational model *while* the computation is in process in a way that supports the interactive manipulation of that representation, two separable but interrelated aspects of visualization procedure must be recognized. First, to evaluate a computational feature through a visual effect, one must establish visualization vocabularies in which the attributes of the representation medium (for present purposes, a two-dimensional color display), e.g., hue and intensity of color, forms (lines, dots, shaded areas), as well as conventional icons (e.g., arrows) are bound, either loosely or rigorously, to specific computational elements. A loosely bound association (e.g., bright red tones to indicate the regions in a contour plot of temperature distribution wherein the highest values fall) operates metaphorically and aesthetically; a rigorous binding is a

[12] BSD is a trademark of the Regents of the University of California.

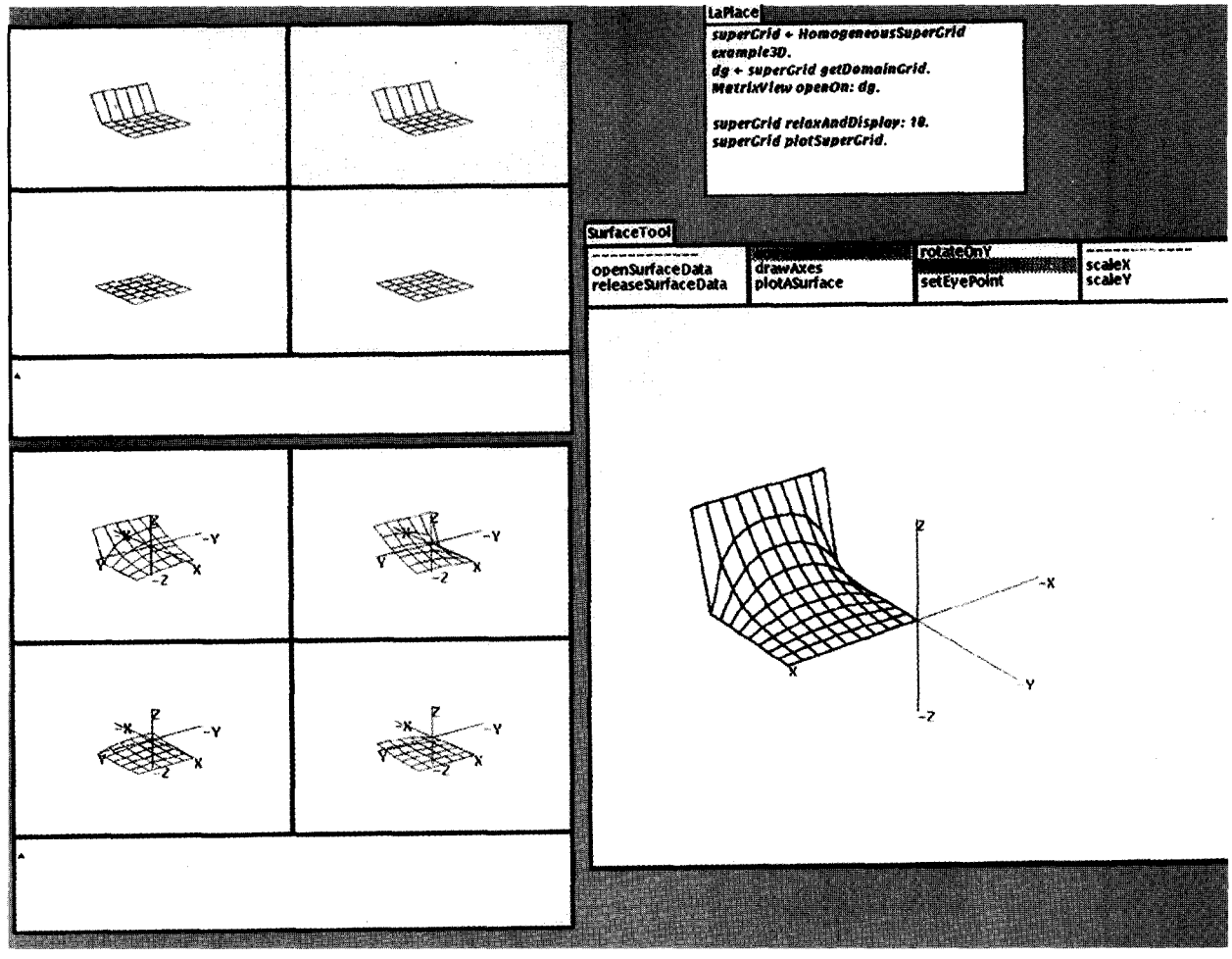

Figure 6

Spreadsheet on a domain decomposition problem. Each cell is computing a region of the problem's physical space. Initial conditions are shown in the graph at top left. The final computation of each subdomain is shown at the bottom left. The solution as reconstituted from the cells can be viewed and manipulated on the tool shown in the window at the right.

rule-governed transformation of a data value to a visual effect at a particular pixel, e.g., the representation of *xy* data values in a two-dimensional plot scaled to its own data ranges and labeled. Such a correlation is frequently thought of as a *denotation*; the visual design points to and is isomorphic with the computational elements.

While we tend to take for granted the associative and rule-governed visualization techniques of the familiar repertoire of scientific graphics, e.g., the Cartesian plots, contour maps, three-dimensional wire-frame surface plots, we are forced to attend to the issue of "graphical meaning" when confronted with a new idiom. For example, apart from their aesthetic appeal, which can be quite powerful, fractal representations provoke recognition of a distinct conceptual framework (the logic of iteration) [7] and yield possibilities for studying iteratively modeled phenomena. Extensions in computational techniques, particularly those involving concurrency, open up wholly new "visualizables" both with respect to the amount of information that can be generated and the dynamics of that generation. The expansion of our visualization vocabulary rests on the collaboration of those who generate and wish to explore their computations and those who design and implement the interfaces through which that exploration is expressed.

Second, when the desired computational information and its visual counterparts have been identified, we face major questions of strategy. Succinctly, it comes down to this: What is the information? When is it available? How

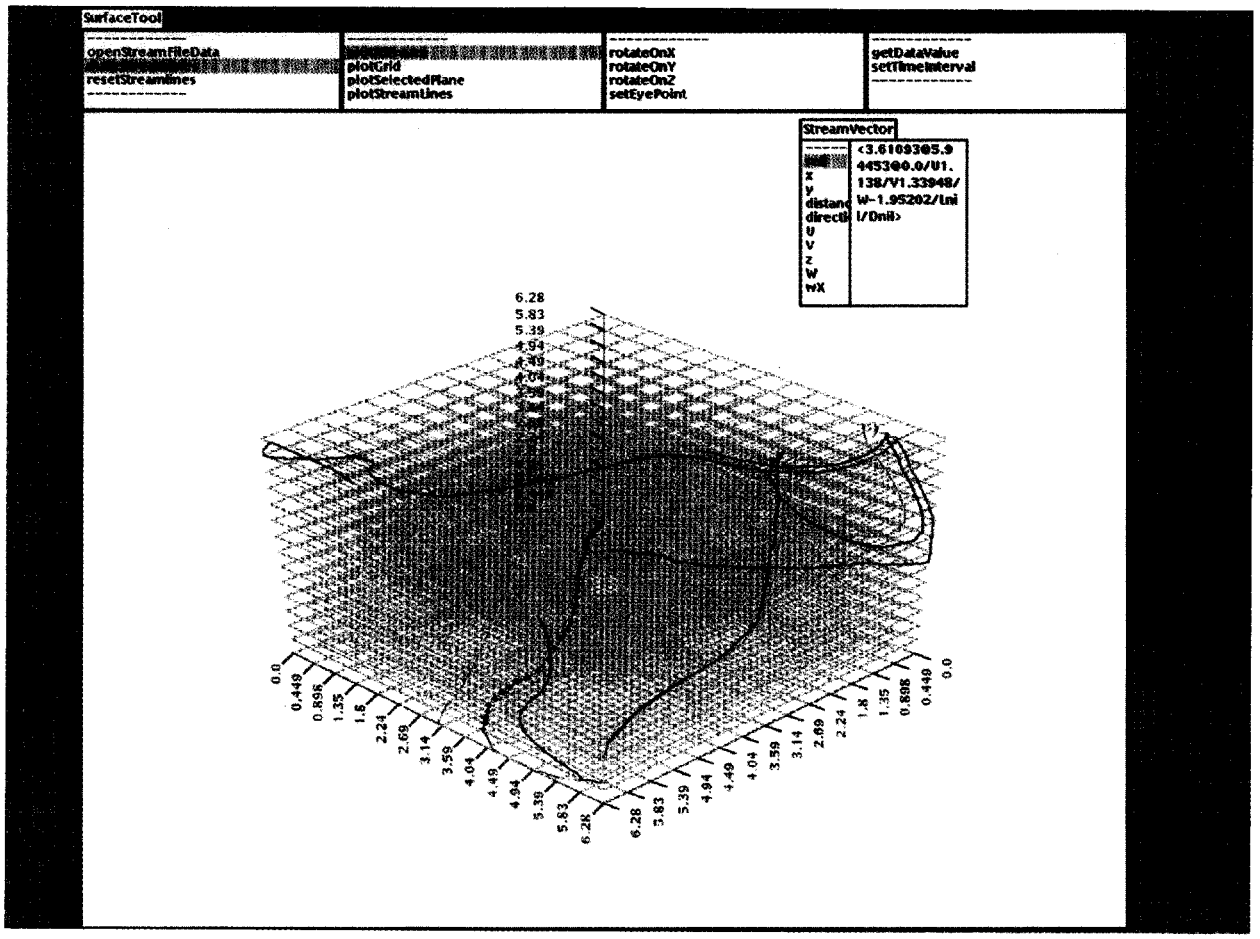

Figure 7

StreamTool on a 15 × 15 × 15 problem. In this view, stream lines have been interpolated from starting points indicated by the user. The entire volume grid and the stream lines can be rotated. A single plane can be selected for display, or the stream lines can be displayed without the spatial gridding.

is it accessed? If incremental visualization is to provide an environment for prototyping genuine scientific problems, it must be capable of managing the volume of data that a numerically stable, physically significant statement of the model would generate. While the maximum capability of virtual on-line real-time data management will increase continually as hardware and software state-of-the-art advances, the data volume we are working to accommodate at this stage would support a three-dimensional fluid-flow computation of a 50 × 50 × 50 matrix across several time slices.

Operationally, the features described above require the following specific capabilities:

1. Retrieval of physical data from display coordinates. The user should be able to indicate via mouse or other pointing device a *screen object*, e.g., a colored pixel, or a region of the screen (a contour section) and thereby cause the retrieval and display of the actual data values that are the basis of the graphic representation and that in fact constitute the *conceptual meaning* of the *perceived* graphic state. For example, Figure 4 shows a flow-field based on a two-dimensional computation in which the user has selected a particular vector (a red dot marks the place where the user positioned the cursor) and an "inspector box" has popped up showing the velocities at the closest meshpoint in the computational grid. Similarly, **Figure 7** shows a three-dimensional computation in which a point on an interpolated stream line has been queried and the values at the closest grid point have been retrieved. The values of the interpolated point could have been retrieved as well.
2. Integration of dynamically generated derivative values into the basic computational information. For example, the user may choose to compute and display

the vorticity at selected positions. These additional data elements are time-consuming to compute and, once computed, need to remain in the computational database but must be integrated into the retrieval strategies so that they can respond to user queries.

3. Organization of historical slices of a computation as a "warehouse" of significant subsections or snapshots of the model as it is computing. The role of the warehouse is to support queries into graphic effects for which the computational trail (or conceptual basis) leads back into stages of the computation that precede the current visualized slice.

All of these requirements can be met by adopting a strategy in which the computational objects whose behaviors we are visualizing are designed as extensible data structures accessible by a search key that is unique for each object, that remains invariant or at least "trackably transmutable" during the computation process, and that lends itself to implementation and manipulation by as few rules as possible. Such a strategy has in fact been employed in the vector visualization tools described above and is best explained by their example.

• *Logical maps and computational objects*

Typically, a computation of a two-dimensional fluid flow yields, as output for graphing, positions and velocities; the positions are the computational grid values or independent variables of each dimension of the computation, and the velocities are those values computed "at" each row/column index of the computational mesh viewed as a two-dimensional matrix. In this case, the computational object of interest, a two-dimensional vector, can be defined as an object whose attributes are a position and a velocity. Furthermore, the attributes themselves are "objects," namely two-dimensional points.

Now, consider a collection of numbers that have been computed by a two-dimensional flow model, using a 32×32 grid. Assume that the numbers are in groups of four, with the order within each group implemented as x, y, u, v. Each group represents a vector; the x and y components represent the position coordinates, the u and v, the velocities. The retrieval keys are the members of the arrays of independent variable points (hereafter referred to as grids) of each dimension of the calculation. The grids can be extracted from the data itself or they can be specified numerically. If the grid indices are given along with the actual data values for the spatial coordinates, the steps that identify the grid values are not needed. This is normally the case when the data have been computed in nonCartesian coordinates. The computational objects themselves can be in any order; all that is required is that a "conceptual element," in this case a vector consisting of position coordinates and velocities, is identifiable as an entity and that it can be parsed into its components.

To extract the grids, the dataset is organized into component collections. In this example, for each vector, its x-position component is placed in a collection of x values and its y-position component in a collection of y values. Each collection is sorted and an *ordered set of unique values* is extracted. If the data are in a Cartesian coordinate system, these sets will represent the grids of each dimension of the computation. The x grid is the ordered set of $x_1, x_2, \cdots, x_n$; the *y grid* is the ordered set of $y_1, y_2, \cdots, y_n$. Since the data collection itself (in effect a collection of records containing x, y, u, v values), can be in any order as long as the internal integrity of each record is maintained, this technique can be used to organize dynamically a collection of data elements that are being computed concurrently but which complete at different times and in indeterminate order.

The indices of the elements in these ordered sets are used as the keys by which any composite element in the dataset can be accessed. Retrieval is based on establishing a mapping of two collections: a numbered list (one-dimensional indexed collection) of the composite data components (in this case, the vectors) in any order, and a mapped collection of n indexors, where n is the number of dimensions of the computation. That is, a two-dimensional computation would establish a map whose entries are referenced by two indexors, i, j; a three-dimensional computation would establish a map whose entries are referenced by three indexors, i, j, k. Each location of the mapped collection (Map[i][j] or Map[i][j][k]) points to the unique data element whose x, y (or x, y, z) values are in fact the grid values at those indices (in the ordered sets). **Figure 8** illustrates these concepts for a small collection of ten objects keyed by a 4×4 mesh.

Although these strategies were first formulated and implemented in Smalltalk, an object-oriented language, they could be implemented in other languages, since the logic does not depend on linguistic features. (In fact, our present system involves concurrent implementation in C *and* in Smalltalk for a distributed computation.) A mapped collection of grid indices to original data is constituted by the following procedures:

```
Let Map[xn][yn] be a Collection [xgrid size] by [ygrid
    size].
Let xorder be the set of xgrid values in ascending order.
Let yorder be the set of ygrid values in ascending order.
```

Map is set up by iterating through the data collection, *finding the index* in the ordered set of grid elements that

Computation objects (vectors)					Ordered grid keys (4 × 4 mesh)			Map (partial) 4 × 4 array keyed as x@y		
index	*x*	*y*	*u*	*v*	*index*	*xorder*	*yorder*	*xindex*	*yindex*	*object index*
1	0.7	0.4	−0.23	0.15	1	0.1	0.2	1	1	9
2	0.3	0.2	−0.44	−0.12	2	0.3	0.4	1	2	3
3	0.1	0.4	0.76	0.00	3	0.5	0.6	1	3	4
4	0.1	0.6	−0.58	0.36	4	0.7	0.8	1	4	nil
5	0.5	0.2	0.63	−0.54				2	1	2
6	0.3	0.4	0.0	0.0				2	2	6
7	0.7	0.8	−0.67	−0.47				2	3	nil
8	0.5	0.4	0.22	−0.68				2	4	10
9	0.1	0.2	0.51	0.17				3	1	5
10	0.3	0.8	−0.79	−0.82				3	2	8

Figure 8

Logical map to a 4 × 4 computation. The left group of items is in arbitrary order. The center shows the *ordered* sets of unique *x* values (xorder) and *y* values (yorder) of the *x* and *y* grids. The right group shows a partial map to the left group based on the center group. The map is shown "partially" for only 10 entries. For a 4 × 4 mesh, the full map would have 16 entries. The example also shows that certain map locations, indexed by valid mesh values, have no computational object and indicate "nil" rather than an index to a computed item.

matches that factor in the composite data element, and using that index as the index for that dimension into the map:

```
1 to: (dataCollection size) do: [: dataIndex
  thisVector := (dataCollection at: dataIndex).
  xindex := match: (thisVector x) to: xorder.
  yindex := match: (thisVector y) to: yorder.
  Map atRow: yindex column: xindex put: dataIndex].
```

Thus, if a given vector, e.g., a vector whose $x = 0.234$ (at xorder [10]) and $y = 0.792$ (at yorder [16]) is at index 321 in the data collection of vectors, Map [10, 16] contains the number 321. With such a map, whenever the *x* and *y* values of a vector can be determined, all other information associated with it can be retrieved. Furthermore, *xy* values not in the original data can be evaluated to find the "closest" or "most appropriate" match in the physical grid. The set of nearest approximations to a match can also be generated. For example, the four nearest neighbors enclosing a point *px py* are found by searching xorder and yorder to find the four grid values "bracketing" *px* and *py*.

These mechanisms underlie the interactive interpolation features in Vectortool that allow the user to select a place in the displayed vector field at which a path line is to be computed. The screen coordinates of the mouse location are "reverse-transformed" to recover the equivalent values in the coordinate system of the data. The "recovered" *x* value is evaluated against the ordered *x* grid to find the closest *x* grid value; the "recovered" *y* value is evaluated against the ordered *y* grid to find the index of the closest *y* value. The resulting indices are used to reach into the map to retrieve the index of the vector object. Retrieving that object from the computational object list provides access to its velocities. By finding the closest grid point and evaluating its position with respect to the selected point, we can retrieve the remaining grid points of the region enclosing the selected point.

This mapping strategy can easily be extended to three-dimensional computations. (Figure 7 shows a volume in which stream lines are traced.) Assume a three-dimensional model whose computational object comprises a coordinate position in 3D space (an *xyz* point) and a velocity in three directions. The organization proceeds as above, with the difference that we now have a *z* grid and another level of indexing into the control map. We have implemented the added dimension by defining a storage object, XYZcollection, as a collection of two-dimensional or XYcollection objects. An XYZcollection stores and retrieves elements by row, column, and depth. The mapping algorithm goes as follows:

```
1 to: (dataCollection size) do: [: dataIndex
  this Vector := (dataCollection at: dataIndex).
  xindex := match: (thisVector x) to: xorder.
  yindex := match: (thisVector y) to: yorder.
  zindex := match: (thisVector z) to: zorder.
  Map atRow: yindex column: xindex depth: zindex put:
  dataIndex].
```

Since each mapped collection location or keyed reference "points" to the corresponding object by storing the *index* of that matched item in the first collection, rather than the item itself, we can extend the data structure of the object itself without having to reconstruct the map. For example, the three-dimensional vector described above could be computed and organized initially as a position and a velocity; its components are six data values representing position (*x*, *y*, *z*) and velocity (*u*, *v*, *w*). Because the key to its retrieval depends entirely on its position components (and their sequential place in the ordered set of mesh values for their dimension), additional data elements can be added to the basic computational objects, extending their storage requirements as well as their definition without disrupting the retrieval mechanism. Thus, a vorticity component for some of the vectors could be computed: By being incorporated into the object collection, it would be immediately "manageable."

The logic of this management strategy is also applicable to adaptive grids (computational mesh regions

that are recalibrated during the course of the computation). In terms of the above description, this means that the *number* of elements in the ordered sets comprising the position keys has changed for a time step or set of time steps. A dataset computed for multiple time steps would be organized as an ordered collection of "computational slices," where each "slice" is a set of computational objects and its map. For nonadaptive grids, the indices of the map would be identical at each time step. Adaptive grids could be accommodated with a *meta map* which would "tag" the indices in a time slice that comprised a subset of, or that had subsumed, other indices.

ODE solution tool

In this section, we describe an interactive, numerical differential equation solver, as an example of some of the IQV ideas presented in this paper. This tool was designed to allow a user to specify an ordinary differential equation (ODE) and visualize the solution interactively, while changing the boundary conditions and the equation parameters. In keeping with the intent of the prototyping environment, the solver is completely automatic, and the working details are transparent to the user (although this can be changed by providing a selection of different solution methods, each of which is more appropriate to a certain equation type). The user need only specify the problem formulation: the equation in string format, the boundary conditions, the discretization resolution, and the solution domain. The parsing of the equation, finite-difference formulation, construction of problem, and solution are all handled by the system.

• *Numerical solution of ODEs*

Numerical solution methods for ODEs are numerous and very well established, and several variations exist. What is needed in a prototyping environment, however, is a robust and quick solution method to facilitate the computational steering step of fine-tuning the design parameters. For this tool, a relaxation method was chosen because it involves computing solutions to large matrix equations which can be implemented on a fast machine in parallel with the user's machine, and the results downloaded, thus making full use of the distributed computing environment. The following is an overview of the steps required in the general relaxation method.

In general, the nth-order differential equation

$$y^{(n)} = f[x, y, y^{(1)}, \cdots, y^{(n-1)}] \tag{1}$$

can be reduced to a set of n first-order equations

$$y_i' = g_i(x, y_1, \cdots, y_n) \qquad i = 1, \cdots, n \tag{2}$$

by using some auxiliary functions [20].

This set of first-order equations is then rewritten as a set of finite difference equations (FDEs) for each of the interior points in the discretized domain. A forward-differencing scheme,

$$u = \frac{1}{2}(u_{p+1} + u_p),$$

$$u' = \frac{1}{\Delta x}(u_{p+1} - u_p),$$

where u_i is the value of the function at point i in the discretized domain, can be used in this case. The FDEs are linearized by rewriting them as a set of linear equations in the highest derivatives, taking the nonlinear terms from the previous iteration. The equations are then expanded into first-order Taylor series with respect to small changes Δy around each of the interior points in the domain. The terms are arranged to form a matrix equation in the correction terms from the series expansion, and the boundary conditions are incorporated to produce an equation of the form

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{3}$$

relating all the interior points in the discretized domain. This equation is more accurately written as

$$\mathbf{A}^{(k-1)} \cdot \mathbf{x}^{(k)} = \mathbf{b}^{(k-1)} \tag{4}$$

and solved several times in a loop, using the solution from the previous iteration $k-1$ to construct the matrix and solve for the next iteration k. The solution is thus relaxed until convergence is reached.

• *Numerical solutions with the ODE tool*

In standard numerical methods, the equations are usually reduced and prepared beforehand, and the computer is only used in the final solution steps that require number-crunching (matrix solutions and relaxation iterations). The purpose of using our prototyping environment is to automate the initial stages as well as the number-crunching steps, so that an equation is processed from string form to a solution plot.

Problem formulation

The most important user-system interface step is the initial one, that of problem formulation. The problem equation must be entered, parsed, and processed. An algebraic manipulator (e.g., Maple [21]) can be used to do this on a remote system, or, as we opted to do, the processing can be done in Smalltalk.

Inheritance rules in Smalltalk allow a subclass to understand the methods associated with its parent class. As mentioned previously, this is very useful for designing special-purpose classes that are very similar to existing classes. By using Smalltalk syntax rules, a string

expression can easily be converted into an equation by using a scanner. The equation $u_{xx} = u$, for example, is entered as 'uxx − u', and when scanned produces a new instance of Array having the value (uxx − u). By analogy, a new subclass of the Smalltalk Scanner class, BVPScanner, was created, and the conversion methods of Scanner were modified for BVPScanner to produce instances of EquationList, a subclass of Array which understands the methods associated with Array (e.g., accessing, adding, removing, testing, etc.), along with all the new methods for algebraic manipulations that have been implemented.

A subclass of the Smalltalk Dictionary class was used to perform this conversion. A set of higher-order derivative variables a, b, c, ⋯) are defined such that

du/dx = ux = a

da/dx = ax = b

⋮

The EquationList instance representing the nth-order equation is recursively scanned and the above substitutions applied, producing n first-order equations. The FDEs are then produced by performing the following substitutions:

ux → (1/deltaX)*((u at : (p + 1)) − (u at : p))

u → 0.5*((u at : (p + 1)) + (u at : p))

ax → (1/deltaX)*((a at : (p + 1)) − (a at : p))

a → 0.5*((a at : (p + 1)) + (a at : p))

⋮

The above chain of translations and substitutions produces equations whose structure may be quite complex, since each substitution adds a level of complexity. As a result, it is necessary to apply a series of expansion and simplification rules to the equations to reduce the equations to a manageable form.

The resulting set of difference equations must be linearized and written in the form given by Equation (4). For this purpose, the variables in the equations must be isolated from their coefficients in the main body of the equations. In a general equation, the highest derivative term in each factor is taken to be the equation variable for step k, while the nonlinear terms are substituted from iteration $k - 1$.

The final result is a template for producing one block in the block-diagonal matrix **A** relating points p and $(p + 1)$. This block is filled in by interpreting it at each interior point on the discretized domain, while the boundary conditions are interpreted at the two boundaries. The resulting equation can be solved through a number of solution schemes in a relaxation loop. We invoke distributed computations at this point and effect the relaxation on a faster remote computer.

Example problem

As an example, the equation $\epsilon u_{xx} - uu_x + u = 0$ with $\epsilon = 0.05$ is processed. The equation is entered in string form as

(0.05*uxx) − (u*ux) + u

The first set of transformations produces two first-order equations and the auxiliary function a:

(ux − a)

((0.05*ax) − (u*a) + u)

Introducing the forward-differentiation scheme produces the following equations:

(((1.0/dX)*(ua0 − us1)) − (0.5*(aa0 + as1)))

((0.05*((1.0/dX)*(aa0 − as1))) −

((0.5*(ua0 + us1))*(0.5 * (aa0 + as1))) +

(0.5*(ua0 + us1)))

In the notation used, the a0 and s1 suffixes denote the function at the points k and $k - 1$, respectively, so that ua0 = (u at: k) and us1 = (u at: (k − 1)). Finally, after expansion and simplification,

(((1.0/dX)*ua0) − ((1.0/dX)*us1) −

0.5*aa0) − (0.5*as1))

((0.05*(1.0/dX)*aa0) − (0.05*(1.0/dX)*as1) −

(0.25*ua0*aa0) − (0.25*ua0*as1) −

(0.25*us1*aa0) − (0.25*us1*as1) +

(0.5*ua0) + (0.5*us1))

Figure 9 shows the matrix block to be interpreted at each point, along with the structure of the final matrix equation. The complete matrix can then be constructed, and the right-hand-side vector **b** determined by evaluating each row in the matrix.

- *The BVP as a computational object*

Although the ODE tool can be used for initial- and final-value problems, the main emphasis is on the two-point boundary-value problem (BVP). A BVP computational object is viewed as a model at the heart of the ODE tool. The object instance variables are listed below:

- *bcs* contains a definition of the boundary conditions. This consists of two lists, one for each boundary. Each list might be nil, or might contain more than one value, corresponding to higher-order boundary

conditions. The order of each value in the list corresponds to the order of the boundary condition it sets.

- *delta* is the discretization resolution of the equation domain.
- *equationList* contains the set of reduced first-order equations.
- *equationString* is the equation as entered in string form.
- *Jacobian* holds the equation template used to build the block-diagonal equation matrix.
- *range* is the discretization domain.
- *resolution* is the discretization resolution.

Using the powerful MVC paradigm, a view and a view controller are associated with this object. These are subclasses of the DataView and DataViewController classes described elsewhere in this paper. **Figure 10** shows the global organization of the tool. As shown in the figure, the BVP model, its view and view controller are maintained by Smalltalk, and the user interfaces with it through the view. Smalltalk, in turn, and independently of the user, can interact with several possible distributed processes to perform any tasks needed by the computations. These can include number-crunching on MATLAB [22] (an interactive version of LINPACK [22]), algebraic manipulation on Maple [21], and database management for providing accurate initial guesses for the solution. These data paths are not necessary, however, if all the processing is done in the Smalltalk environment.

Creating the BVP model

The model is created either by explicitly stating all the parameters, then opening the view,

```
aBVP := TwoPointBVP
            newFromEquation:
            withBCs:
            resolution:
            range:
aBVP openview.
```

or by opening a view on a nil model

```
TwoPointBVP start.
```

and using the *newBVP* option to fill the model's parameters. When creating the object, Smalltalk also opens data paths to MATLAB, Maple, and any other remote system, and initializes the problem on these systems.

Viewing the BVP solution

The BVP view (an instance of TwoPointBVPView) is shown in **Figure 11**. As shown, there are three option lists and two slide gauges. The option lists allow the user to send computation messages to the model, while the slide

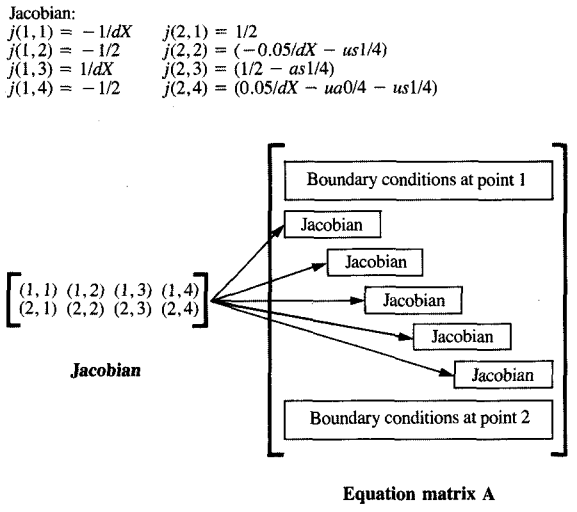

Figure 9

Building the equation matrix **A**. The Jacobian obtained from the Taylor expansion of the finite difference equations is used as a building block to relate points k and $k + 1$ in the internal discretization of the solution domain. The boundary conditions are handled separately.

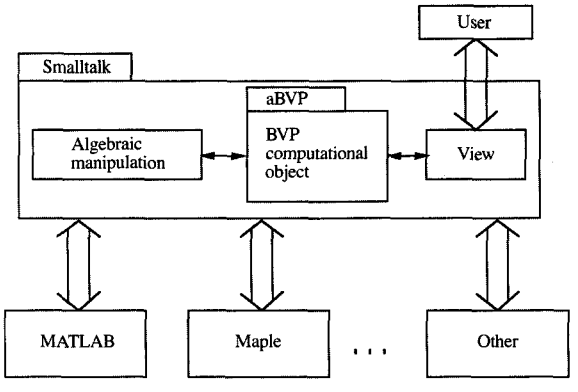

Figure 10

Global organization of the ODE tool. The BVP object is maintained as the model of an MVC structure in Smalltalk. The user interacts with this model through its view and view controller. The algebraic manipulator and any other Smalltalk applications send messages directly to the model, whereas Smalltalk interacts with remote applications as needed through separate data paths, independently of the user.

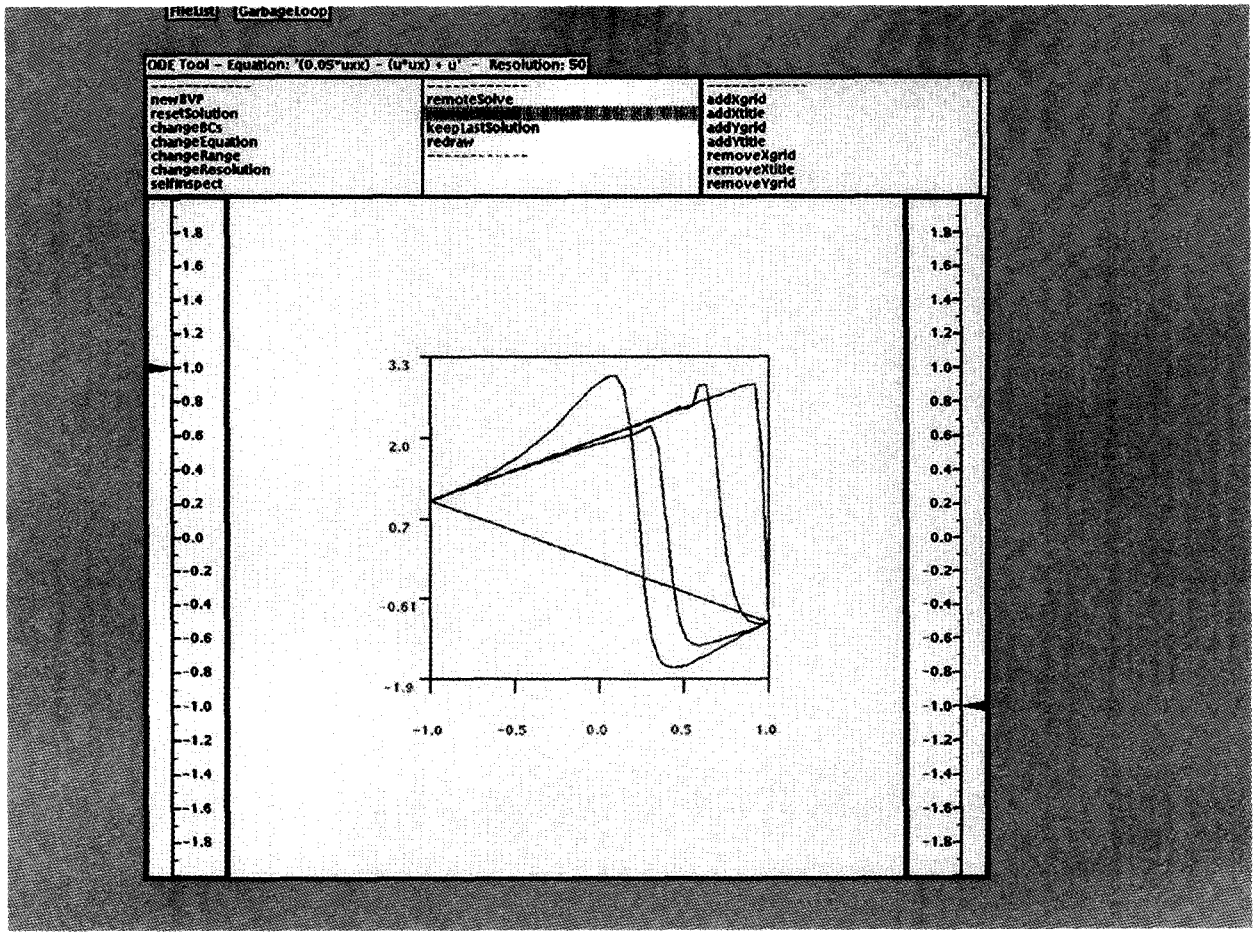

Figure 11

ODE tool—relaxation steps. The progress of the relaxation toward the solution of the shock equation $\epsilon u_{xx} - uu_x + u = 0$ with $\epsilon = 0.05$ is shown. The initial guess is the line between the boundary conditions. The steps shown are five iterations apart.

gauges are used to modify the boundary conditions. The first option list is concerned with setting the various parameters of the equation. The possible options are listed below:

- *newBVP* Start up another problem. The user is led through the steps of specifying the complete problem. This is one of the two possible ways to create a BVP.
- *resetSolution* The displayed solution is reset to the initial guess, which is usually a line between the boundary conditions, unless an initial guess has been obtained from a knowledge base [23].
- *changeBCs* Change the boundary conditions. This option can be used in addition to the slide gauges.
- *changeEquation* Change only the equation for the problem. All the other specifications (boundary conditions, resolution, range) are kept. The user can also specify that the displayed solution be kept as an initial guess for the next problem. This feature is extremely useful when successive approximations for an equation parameter are investigated. The equation can be solved with the first parameter, the parameter changed while keeping the last obtained solution, and the process repeated.
- *changeDomain* Change the problem domain, keeping the same resolution.
- *changeResolution* Change the discretization resolution.
- *acceptGauges* Accept the input from the sliding gauges. The gauges are used to enter the boundary conditions for the TwoPointBVP.
- *selfInspect* Instruct Smalltalk to open an Inspector window on the model.

The options in the second window handle the visualization process. The options are the following:

- *remoteSolve* Send the 'solve' command to the remote system. The command activates a command file that repeats the iteration step until convergence to a tolerance set by the user is reached. The solution is then displayed and previous results erased.
- *remoteIterate* Send the 'iterate' command to the remote system. This option allows the user to perform a limited number of iteration steps and display the intermediary results. Previous results are displayed superimposed on the same plot.
- *keepLastSolution* All the results except the last one are erased.
- *redraw* The plot is redrawn.

The other window is concerned with the visual presentation of the x and y axes, ticks and grids on the plots. They are inherited from the DataView class.

Example application

The ODE tool was found to be very useful in exploring the behavior of differential equations, especially nonlinear equations, by allowing the rapid visualization of the solutions as the parameters and boundary conditions were changed. To illustrate this process of creating and solving an equation, we go through the steps for the equation shown in **Figure 12**. The problem specification is entered from the workspace as

```
aBVP := TwoPointBVP
        newFromEquation: (0.1*uxx) − (u*ux) + u
        withBCs: ((1.0) (−1.0))
        resolution: 40
        range: (−1.0 1.0).
aBVP openview.
```

From the view, *remoteIterate* is first selected for two iteration steps to check for the stability of the solution, then *remoteSolve* to proceed with the solution [Figure 12(a)]. The parameter is changed now from 0.1 to 0.05 by selecting *changeEquation* and choosing to keep the old solution as a starting guess for the new one, thereby reducing the computation time [Figure 12(b)]. Now, the parameter is further reduced to 0.01 with *changeEquation*. This time, however, the iterations from *remoteIterate* show some instability in the solution [Figure 12(c)], which leads us to increase the discretization resolution by selecting *changeResolution* from 40 to 200. This again results in a stable solution [Figure 12(d)].

Flag simulation

The applicability of the Smalltalk-80 IQV environment to the development of numerically intensive models was tested on a flag-simulation model. This problem was chosen because of its complexity of implementation and computation, particularly as a means of testing the Smalltalk/IQV paradigm; the motion of the flag occurs in three dimensions and requires three-dimensional vector algebra for the computation. In addition, the appearance of a flag moving in the wind is familiar to most people. The flag model essentially consists of a grid of mass points connected by a set of springs. This sheet of masses and springs is fixed in space at two points, P1 and P2, as depicted in **Figure 13**. The flag is subject to a net force, $\mathbf{F}_{net}$, which is made up of four forces:

$$\mathbf{F}_{net} = \mathbf{F}_{spring} + \mathbf{F}_{friction} + \mathbf{F}_{grav} + \mathbf{F}_{wind} .$$

The four forces are the spring, frictional, gravitational, and wind forces. These forces on an arbitrary mass point are given by the vector relations

$$\mathbf{F}_s = \sum_{i=1}^{8} k\, \Delta\mathbf{x}_i ,$$

$$\mathbf{F}_f = r\, \mathbf{v}_0 ,$$

$$\mathbf{F}_g = g\, \mathbf{e}_y ,$$

$$\mathbf{F}_w = w[(\mathbf{v}_0 - \mathbf{v}_w) \cdot \mathbf{n}_0]\mathbf{n}_0 .$$

In the equations above, boldface denotes vectors, $\Delta\mathbf{x}_i$ represents a vector to the ith neighbor, $\mathbf{v}_0$ represents a velocity vector, $\mathbf{e}_y$ represents the unit y-direction vector, and $\mathbf{n}_0$ represents the unit normal to the flag at a given mass point. The relative strengths of the forces are given by the parameters k, r, g, and w. The position and velocity vectors, $\mathbf{x}_0$ and $\mathbf{v}_0$, at each mass point are computed at time $t + \Delta t$ from time t by applying the following kinematic relations:

$$\mathbf{x}_0(t + \Delta t) = \mathbf{x}_0(t) + \mathbf{v}_0(t) \cdot \Delta t + 0.5 \cdot \mathbf{a} \cdot \Delta t^2,$$

$$\mathbf{v}_0(t + \Delta t) = \mathbf{v}_0(t) + \mathbf{a} \cdot \Delta t,$$

where

$$\mathbf{a} = \mathbf{F}_{net}/m.$$

The flag simulation consists of multiple repetitions of update and display, using the velocity and position update formulas above.

Flag IQV computational objects

The definition of usable objects in the IQV environment led to quick prototyping of the flag simulation. Code development consisted of creating structures which encapsulated much of the computational detail, thus allowing a higher-level "language" for programming. New classes were built incrementally from existing classes; each new class addressed the flag simulation at a higher level of abstraction. The ease of visual display at any level

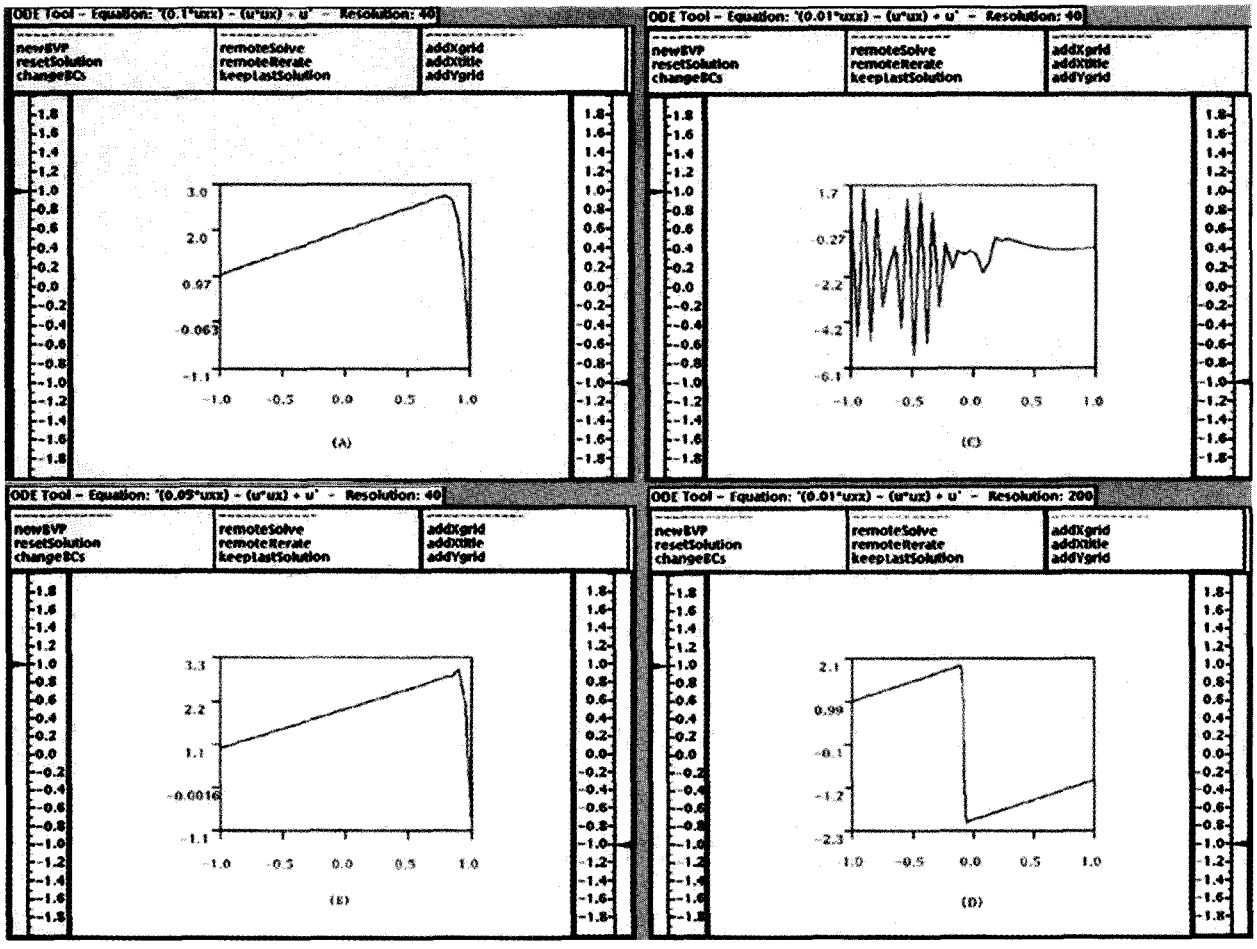

Figure 12

A session with the ODE tool. The BVP computational object for the same shock equation with $\epsilon = 0.1$ was created in a workspace (see text). The parameter ϵ is varied to determine its effect on the final solution:
(a) For a value of $\epsilon = 0.1$ and a resolution of 40, *remoteSolve* yields a stable solution.
(b) Changing ϵ to 0.05 still yields a similar solution.
(c) When ϵ is further changed to 0.01, the coarse resolution causes instabilities in the solution, shown by performing one iteration with *remoteIterate*.
(d) Changing the resolution to 200 points with *changeResolution* leads to a stable solution again.

of abstraction contributed greatly to the programming efficiency. This section demonstrates this idea.

The first task consisted of defining a vector class in order to deal with the vector algebra. This stage of programming led to the programmer's ability to perform the following *types* of operations:

a := Vector3D new.	"Create a new 3D vector."
b cross: a.	"Vector cross product."
c*(1/2).	"Multiplying vector by scalar."
d norm.	"Length of vector d."

As is evident from this code, there was no need to deal with the details of vector operations from that point onward.

Next, the classes Flag and FlagPoint were developed. These classes allowed the forces to be dealt with at a very high level. For instance, the four flag forces described above were each applied to all the mass points (FlagPoint objects) in the model. The classes provided easy access to the flag data, as well as a logical, simple command syntax. For example, an instance of Flag could be invoked and displayed by the following series of Smalltalk commands:

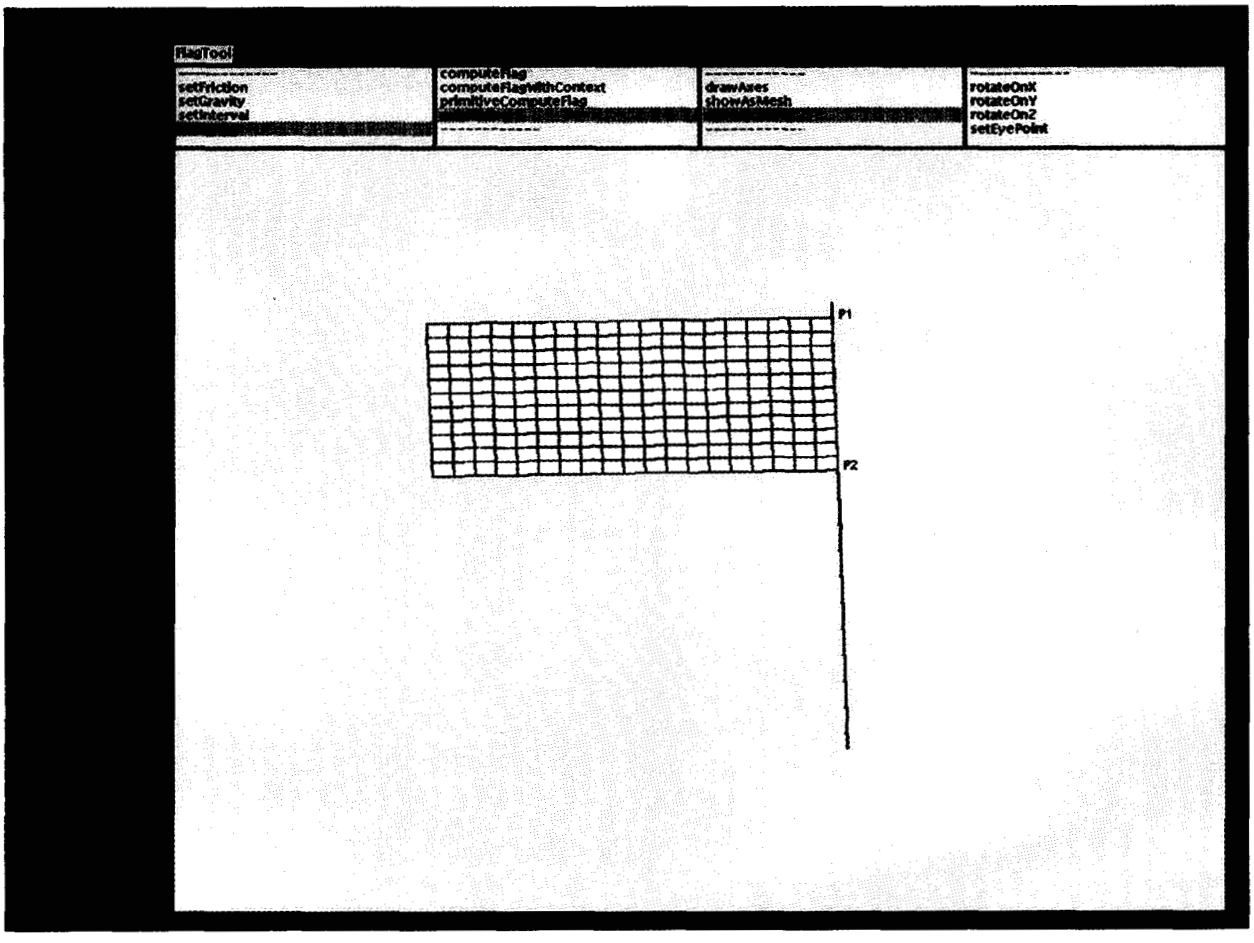

Figure 13

FlagTool simulation environment.

```
flag := Flag new.        "Create a flag model."
flag initialize.         "Initialize the flag position."
flag openFlag.           "Open the IQV environment."
```

Immediate visualization of the modified forces could be obtained in this fashion. **Figure 14** depicts the Flag environment and the current version of the frictional force component procedure. The flag could be made to flap for several iterations, then visually debugged in the Flag environment. The appropriate force terms could then easily be edited. The flag would, thereafter, resume its motion with the corrected force term applied.

Several parameters required subtle modifications in order to ensure that the flag model was visually realistic. For instance, the wind velocity, the spring constant, the frictional coefficient, and the downward pull on the flag all required adjustment. This portion of code development is called *physical debugging*. If, for example, the frictional coefficient was made too low, the flag motion appeared much too random. The IQV environment allowed the feedback process between the numerical model and the programmer to proceed quickly enough that a change could be incorporated into the flag environment in just a few seconds. The resulting model (**Figure 15**) would then proceed to flap, this time with a higher frictional coefficient, and in a more realistic visual manner.

The IQV environment certainly enabled a manyfold increase in programming speed in this instance. The rapid feedback to the user was particularly useful in the physical debugging stage of simulation.

Conclusions and future research

In this paper we have described strategies to effect interactive graphical support for the quantitative requirements associated with prototyping of scientific and

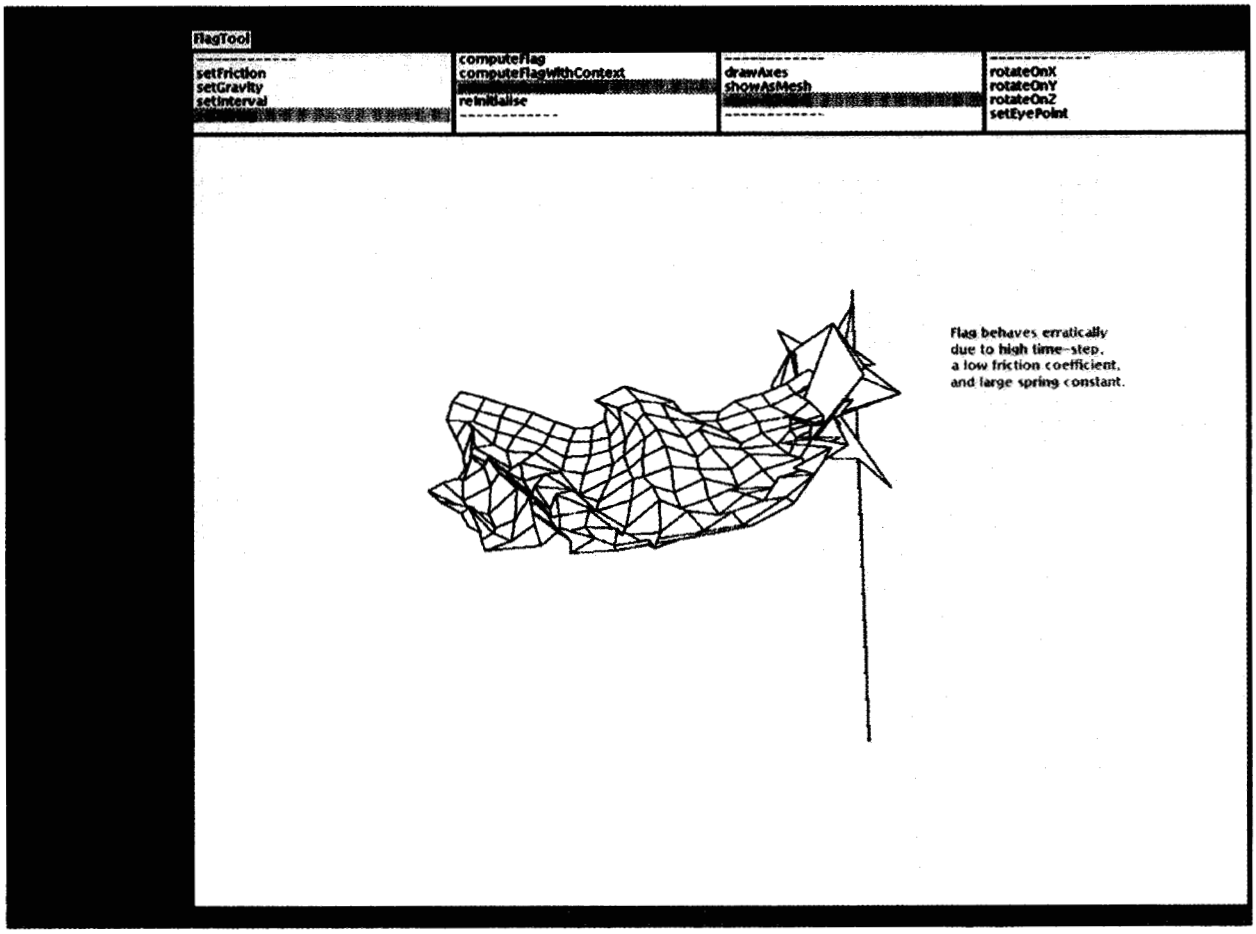

Figure 14

FlagTool showing simulated flag. This view shows erratic flag motion due to improper setting of friction, spring, and time-step parameters.

engineering computer simulations. Object-oriented software and access to distributed computing resources are important features of our system. Flexible graphics tools which emphasize quantitative connection with the underlying computational data are essential. The user's ability to recover his computed data from any arbitrary graphical view depends on a dictionary-based data structure which serves as a link between the graphical visualization and the real data. With this implementation, the user can interactively query any visual point for its associated datasets.

In the examples presented, we have shown how IQV supports our concept of computational steering. The system constructed for nonlinear boundary-value problem solution also illustrates how distributed computer resources, accessible to the user in a transparent manner, are employed to improve performance, which itself is an important aspect of the interactive environment. Both the differential equation solver system and the flag model show how IQV is used to support incremental *in situ* changes to the mathematical model. In our system, the implementation of computational steering depends on Smalltalk's incremental compilation capability.

We have established the feasibility of new forms of computer environments for the scientist and engineer; it is no longer necessary for these professionals to be constrained to old *edit-compile-link-run* environments with only postprocessing graphics as an option. The scientific professional needs the same modern interfaces we now afford businessmen who use small computers. Much remains to be done before these new scientific environments will be accepted into widespread use.

There is a need to replace the "string" format used for equation input with the capability for input expression in mathematical symbols; this is a nontrivial task in

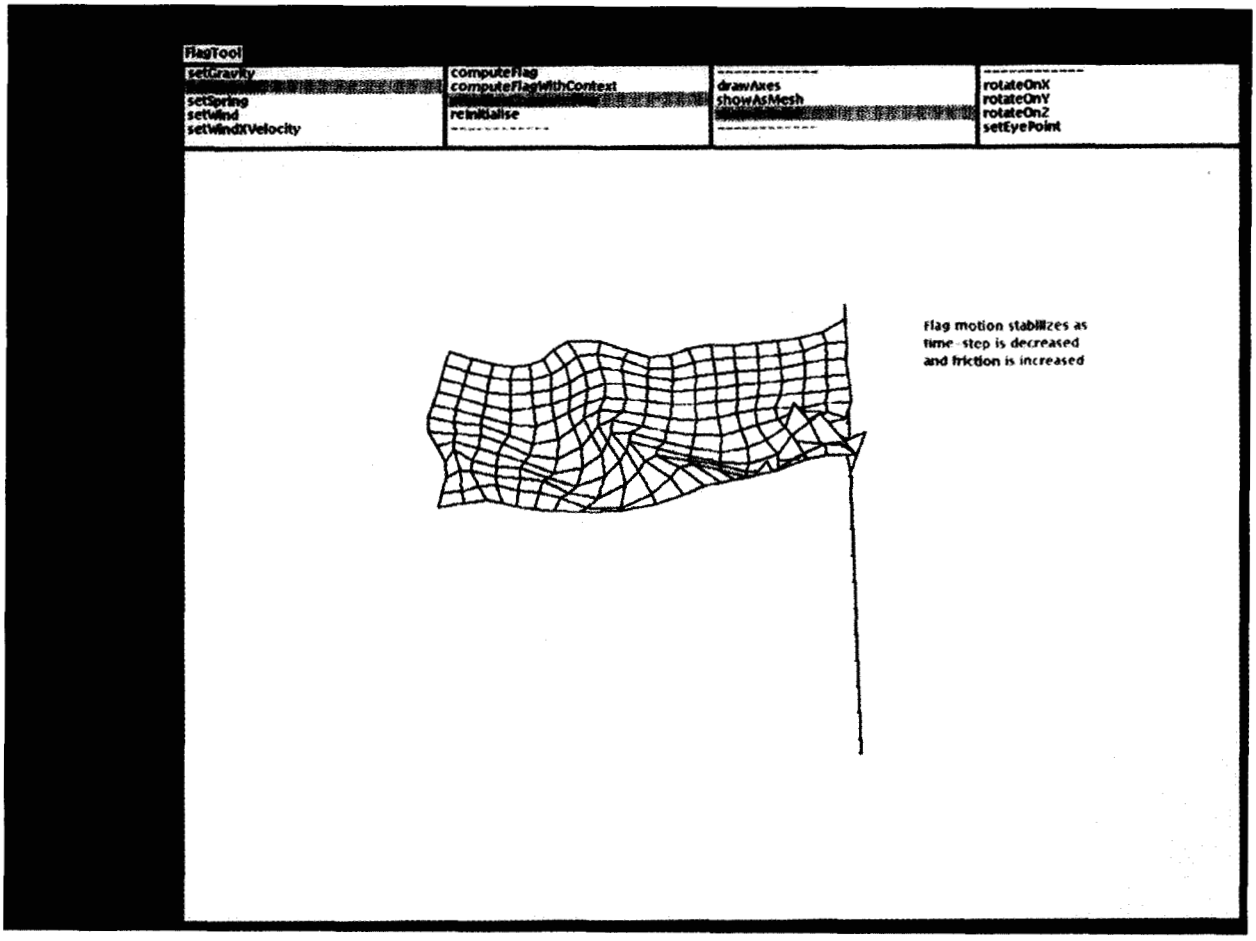

Figure 15

FlagTool showing improved simulation. This view shows stabilized flag motion following the tuning of friction, spring, and time-step parameters.

contextual pattern recognition. Intelligent databases (e.g., expert systems) need to be added to tools like our boundary-value problem solver, so that these tools can have access to theoretical information such as methods for high-order approximations. Inclusion of such access will allow faster and more accurate prototype solutions. While we use distributed computing, there is a need to develop internal system "intelligence" so that the most appropriate resource is used. Additionally, parallel and concurrent computation must become a more integral part of the overall environment, particularly the parts that support interactive graphics. As our graphical requests become more complex, parallel computation will be required to implement database access, perform transformations, and service user requests in a deterministic manner. Perhaps the most challenging task is to integrate visual "realism," with all of its complex rendering, and IQV so that users can have access to as complex and dynamic graphical representation as they need, without losing interactive connection to their physical model and its data.

Acknowledgments

This research was supported in part by the National Science Foundation under NSF Grant ECS-8814937, and by the Computational Engineering Systems Laboratory of the Center for Computer Aids for Industrial Productivity (CAIP). CAIP is supported by the New Jersey Commission on Science and Technology, Rutgers—the State University of New Jersey, and the CAIP Industrial Members.

References

1. B. McCormick, T. DeFanti, and M. Brown, "Visualization in Scientific Computing," NSF Grant ASC-8712231, National Science Foundation, July 1987.

2. *Products Catalog*, Tektronix Corporation, Beaverton, OR, 1989.
3. R. A. Dory, "Wingz as a Scientific Aid," *Computers in Phys.* **3,** No. 4, 93–100 (1989).
4. F. J. Bitz and N. J. Zabusky, "David and 'Visiometrics': Visualizing and Quantifying Evolving Amorphous Objects," *Computers in Phys.* **4,** No. 6, 603–613 (1990).
5. B. Bacon, "Visualization at SIGGRAPH," *Computers in Phys.* **3,** No. 5, 11 (1989).
6. T. Diede, C. F. Hagenmaier, G. S. Miranker, J. R. Rubenstein, and W. S. Worley, "The Titan Graphics Supercomputer Architecture," *IEEE Computer* **21,** No. 9, 13–30 (1989).
7. D. Rapaport, "Visualizing Physics," *Computers in Phys.* **3,** No. 5, 18–29 (1989).
8. J. Bourne, J. Cantwell, A. Brodersen, B. Antao, A. Koussis, and Y. Huang, "Intelligent Hypertutoring in Engineering," *Academ. Computing* **4,** No. 1, 18 (1989).
9. S. Walther, "Strategies for Interactive Graphing of Numeric Results," *Proceedings of the International Symposium on AI, Expert Systems and Languages in Modelling and Simulation (IMACS)*, Barcelona, Spain, June 1987, pp. 379–384.
10. R. L. Peskin, S. S. Walther, and A. M. Froncioni, "Smalltalk—The Next Generation Scientific Computing Interface?" *Math. & Computers in Simulation* **31,** No. 4, 5, 371–381 (1989).
11. S. S. Walther and R. L. Peskin, "Strategies for Scientific Prototyping in Smalltalk," *Proceedings of the Fourth Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA/ACM)*, October 1989; **24,** No. 10, 159–169 (1989).
12. S. Knaster, *Macintosh Programming Secrets*, Addison-Wesley Publishing Co., Reading, MA, 1988.
13. A. Goldberg and D. Robson, *Smalltalk-80*, Addison-Wesley Publishing Co., Reading, MA, 1983.
14. C. Chambers, D. M. Ungar, and E. Lee, "An Efficient Implementation of SELF, a Dynamically Typed Object-Oriented Language Based on Prototypes," *Proc. OOPSLA/ACM* **24,** No. 10, 49–70 (1989).
15. B. Foote and R. E. Johnson, "Reflective Facilities in Smalltalk-80," *Proc. OOPSLA/ACM* **24,** No. 10, 327–337 (1989).
16. N. Carriero and D. Gelernter, "Linda in Context," *Commun. ACM* **32,** No. 4, 444 (1989).
17. S. S. Walther, "Interactive Visualization and Control of Parallel Computations," *Proceedings of the Third Annual User-System Interface Conference (USICON)*, Austin, TX, February 1988, pp. 52–59.
18. P. Berge, V. Pomean, and C. Vidal, *Order Within Chaos*, Hermann, Paris, France, 1984.
19. S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Co., Reading, MA, 1989.
20. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, England, 1986.
21. *Maple Reference Manual, 5th Ed.*, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1988.
22. T. F. Coleman and C. V. Loan, *Handbook for Matrix Computations*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1988.
23. Mark F. Russo, "Automatic Generation of Parallel Programs Using Nonlinear Perturbation Theory," Ph.D. Dissertation, Rutgers, The State University of New Jersey, New Brunswick, NJ, 1989.

Received November 16, 1989; accepted for publication September 17, 1990

Richard L. Peskin *Center for Computer Aids for Industrial Productivity (CAIP), P.O. Box 1390, Rutgers University, Piscataway, New Jersey 08855.* Dr. Peskin is Professor of Mechanical and Aerospace Engineering and Director of the Computational Engineering Systems Laboratory of the Center for Computer Aids for Industrial Productivity (CAIP) at Rutgers University. He holds a B.S. from the Massachusetts Institute of Technology and an M.S.E., an M.A., and a Ph.D. from Princeton. He is a member of IMACS, ASME, APS, and other professional societies. Dr. Peskin has published extensively in the fields of computational fluid dynamics, combustion, atmospheric science, and computer applications. His current research interests include parallel computing, symbolic computing, interactive interfaces for scientific computation, and nonlinear dynamics.

Sandra S. Walther *Center for Computer Aids for Industrial Productivity (CAIP), P.O. Box 1390, Rutgers University, Piscataway, New Jersey 08855.* Dr. Walther is Project Leader of the Graphical Interfaces and Data Management group in the Computational Engineering Systems Laboratory of the Center for Computer Aids for Industrial Productivity (CAIP) at Rutgers University. She received her B.A. in general studies in the humanities from the University of Chicago and her M.A. and Ph.D. in philosophy from Yale University, with concentration in logic and philosophy of science. Dr. Walther's current research centers on object-oriented interfaces to parallel and distributed scientific computations and has resulted in prototype software systems in use by academic and industrial researchers.

Andy M. Froncioni *Department of Mechanical and Aerospace Engineering, Rutgers University, Piscataway, New Jersey 08855.* Mr. Froncioni holds B.Sc. (physics), B.Eng. (electrical), and M.Eng. (electrical) degrees from McGill University. He is currently a Ph.D. student in the Mechanical and Aerospace Engineering Department at Rutgers University. Mr. Froncioni has published in the fields of numerical analysis and object-oriented programming. His current interests include computational fluid dynamics and automatic parallel code generation.

Toufic I. Boubez *Biomedical Engineering Department, Rutgers University, Piscataway, New Jersey 08855.* Mr. Boubez holds B.Eng. and M.Eng. degrees from McGill University and is currently a Ph.D. student in the Biomedical Engineering Department at Rutgers University. He has published in the fields of automatic mesh generation, numerical methods, and object-oriented programming. His current research interests include neural networks, object-oriented programming, parallel computing, and scientific visualization.