Field Model: An Object-Oriented Data Model for Fields

Technical Report NAS-01-005

Patrick J. Moran*
NASA Ames Research Center
pmoran@nas.nasa.gov

Abstract

We present an extensible, object-oriented data model designed for field data entitled *Field Model* (*FM*). *FM* objects can represent a wide variety of fields, including fields of arbitrary dimension and node type. *FM* can also handle time-series data. *FM* achieves generality through carefully selected topological primitives and through an implementation that leverages the potential of templated C++. *FM* supports fields where the nodes values are paired with any cell type. Thus *FM* can represent data where the field nodes are paired with the vertices ("vertex-centered" data), fields where the nodes are paired with the D-dimensional cells in $\mathbf{R}^D$ (often called "cell-centered" data), as well as fields where nodes are paired with edges or other cell types. *FM* is designed to effectively handle very large data sets; in particular *FM* employs a demand-driven evaluation strategy that works especially well with large field data. Finally, the interfaces developed for *FM* have the potential to effectively abstract field data based on adaptive meshes. We present initial results with a triangular adaptive grid in $\mathbf{R}^2$ and discuss how the same design abstractions would work equally well with other adaptive-grid variations, including meshes in $\mathbf{R}^3$.

CR Categories: E. Data (large); I.1.3 Languages and Systems, Evaluation strategies; I.3.8 Computer Graphics Applications

Keywords: data models, object-oriented, C++, templates, scientific visualization, demand-driven evaluation.

1 Introduction

Underlying virtually every object-oriented visualization system is a data model. The data model forms a key part of the system design, effectively spelling out the types of data that can be analyzed by the system. A well-designed data model component can significantly enhance the capabilities of the overall system. For example, the developers of OpenDX (formerly IBM Data Explorer) often cite the consistent, unified nature of the DX data model as one of the key reasons for the success of their system [13, 1]. For large data visualization, the data model can have a significant impact on system efficacy. Poorly chosen abstractions can lead to performance problems or make development awkward. Well-designed abstractions can enhance code reuse and enable the coupling of components in new and interesting ways.

A recent trend in numerical computing is the growing popularity of adaptive meshes. Adaptive meshes increase or decrease resolution automatically as required by a simulation code. Adaptive meshes free the scientist from having to construct a mesh initially that completely anticipates where high resolution will be required. Adaptive meshes are also a natural choice when the resolution required in various regions of the domain changes over the course of the simulation, for instance, following a shock wave. Adaptive mesh techniques are often implemented as parallel algorithms, requiring careful load balancing and communication strategies in order to be most effective. Unfortunately, adaptive meshes tend not to match well with the data models underlying current general visualization systems, prompting mesh library developers to resort to developing visualization modules custom to their mesh design.

For those in the visualization community, adaptive meshes offer the possibility of new and interesting research topics. For example, one might want to couple various multi-resolution visualization techniques with the adaptive mesh data structures. For visualization system developers, adaptive meshes are a challenge. There are a number of current adaptive mesh development efforts, each with its own custom algorithms and data structures. One would like to apply the wealth of visualization techniques that have already been developed, yet one is likely not to have the resources to devote to interfacing to each adaptive mesh variation. This is where a carefully designed data model comes in. With appropriately chosen abstractions, a data model can insulate the visualization techniques from the majority of the idiosyncrasies of the mesh and field data structures. A carefully designed model can also enhance modularity: newly added mesh and field types in the future should not require significant modifications to existing code.

In general, the advantages of a good data model are not limited to adaptive meshes alone. Overall, our goal is to provide a common model for field data that will enhance the sharing of data sets and of visualization technique implementations. In the next section we provide an overview of some of the key concepts in the *FM* design that are intended to take us towards our goal. Following that we survey related data model work within the visualization community. Next, we discuss key features of the *FM* design, and then present current results. Finally, we conclude with a discussion of future plans for the *FM* project.

2 Field Model Concepts

Field Model objects are embedded in $\mathbf{R}^D$, also known as *physical space*. Objects in $\mathbf{R}^D$ are also said to have a *physical dimensionality* of D. The regions in $\mathbf{R}^D$ where fields are defined are discretized by meshes, which in turn are composed of cells. A *k-cell* is a subset of $\mathbf{R}^D$ that is homeomorphic (topologically equivalent) to a k-ball. Cells in *FM* are currently all linear objects. A 0-cell is a vertex, a 1-cell is an edge, 2-cells include triangles, quadrilaterals, and other polygons. Hexahedra, tetrahedra, pyramids and prisms are all examples of 3-cells. Every cell σ has a set of vertices. We use a more general concept of face than some are familiar with: a *face* of σ is specified by a non-empty subset of the vertices of σ [1]. For example, a hexahedron has not only quadrilateral faces, but also vertex and edge faces. Every cell is also a face of itself. The general face definition enables us to develop a more uniform treatment of objects

*Mail Stop T27A-2, Moffett Field, CA 94035

[1] If a cell σ is not a simplex, then not every subset of the vertices of σ constitutes a face. In practice it is clear which subsets define valid faces.

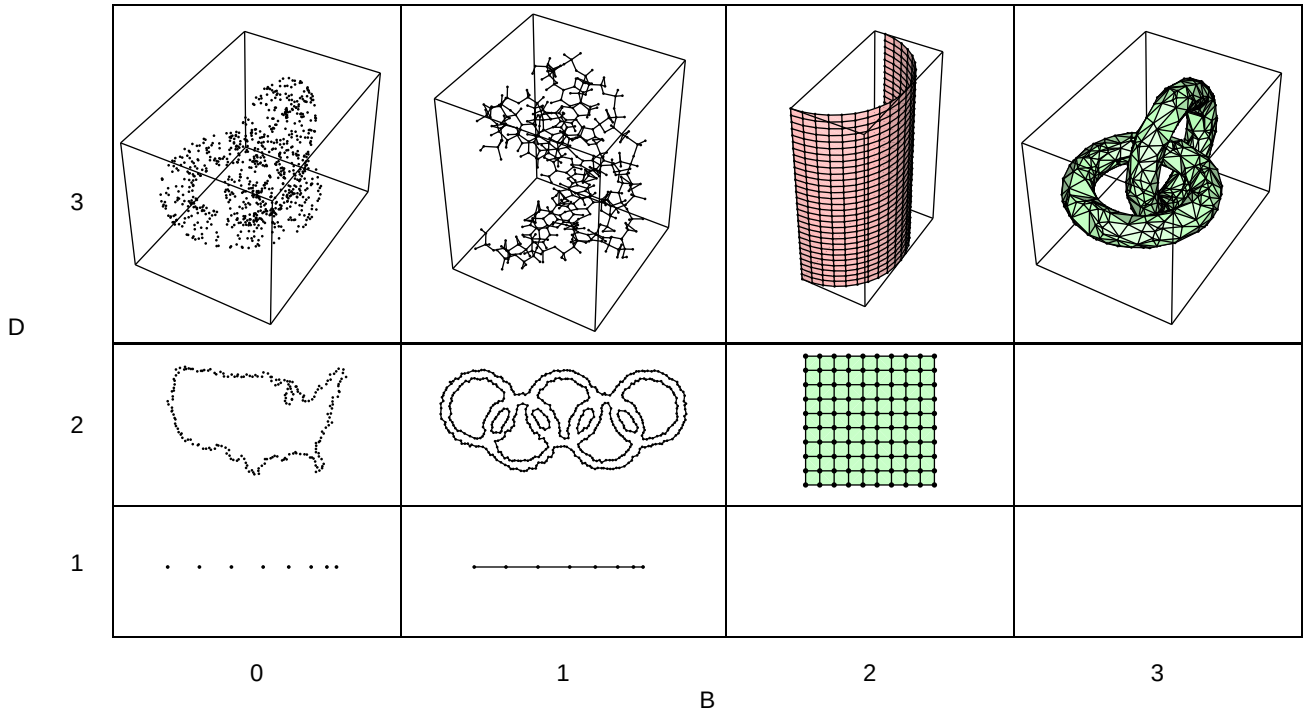


Figure 1: Example *FM* meshes organized in columns and rows by base dimensionality (B) and physical dimensionality (D), respectively. Note that the model is general enough to represent not only the input to visualization techniques, but also the output. For example, regular meshes in $\mathbf{R}^2$ ((B,D) = (2,2)) could serve as the underlying discretization for images, and surfaces in $\mathbf{R}^3$ naturally correspond to (2,3) meshes.

with general dimension. A *mesh* $\mathcal{M}$ is a finite set of cells such that if $\sigma \in \mathcal{M}$, and τ is a face of σ, then $\tau \in \mathcal{M}$. Typically, cells in a mesh share common faces, so for example two tetrahedra can share triangle, edge, and vertex faces. If the cells with the highest dimensionality in mesh $\mathcal{M}$ are B-cells, then $\mathcal{M}$ is a B-mesh, and $\mathcal{M}$ has a *base dimensionality* of B. The base dimensionality of a mesh must be less than or equal to its physical dimensionality. The *shape* of a mesh $\mathcal{M}$ is the space occupied by the union of all the cells of $\mathcal{M}$. In most cases, the shape of a B-mesh is equivalent to a B-manifold with boundary. In order to rule out some cell collections that do not have a manifold shape, we require that every cell in a B-mesh $\mathcal{M}$ must be the face of some B-cell in $\mathcal{M}$. This requirement, for instance, rules out cases where we have a surface ($B = 2$) with spurious edges and vertices that are not part of the surface.

Figure 1 illustrates example meshes that can be constructed in *FM*. Note that *FM* meshes can represent familiar objects such as regular meshes, curvilinear meshes, and tetrahedral unstructured meshes. Note too that our definition is general enough that we can represent objects less commonly thought of as meshes, such as a collection of vertices and edges signifying the atoms and bonds of a molecule ($B = 1, D = 3$). Also, note that the molecular example is a case where the set of cells adheres to our mesh definition, but the shape of the mesh is non-manifold.

A *field* defines a function within a region of space. In *FM*, each field object has a set of values called *nodes* (which can be accessed on demand), a mesh, and a pairing between the k-cells in the mesh and the nodes. The value of k for a particular field is known as its *node association index*. The base and physical dimensionalities of a field are the dimensionalities of its underlying mesh. For fields with base dimensionality B, the most common node association indices seen in visualization data are 0 ("vertex centered") and B (typically called "cell centered"). Other node association indices tend to be underrepresented in visualization studies, though they are still important scientifically. Node association index 1 fields often occur in electromagnetics simulations as well as some adaptive mesh systems, where adaptation criteria are paired with the edges. Node association index 2 fields are useful in some flow studies, where fluxes are tracked at the 2-cells in order to verify the correctness of the simulation.

For a field with node association index k, the user can request a single value at a particular k-cell or request multiple values at a j-cell, $j \neq k$. We define later how the field selects node values in the case where $j \neq k$. The user can also request a field value at an arbitrary point in physical space, or for fields based on meshes with structured behavior, at an arbitrary point in base space. In response to such queries, fields return an integer code indicating whether the query was successful (e.g., depending upon whether the given point was within the part of the domain where the field is defined), and a field value. Appropriate interpolation techniques are fairly well agreed upon for fields with node association index 0; for other node association indices appropriate interpolation methods are still under investigation.

Before proceeding with a description of the *FM* design and implementation, we review previous data model work.

3 Related Work

The importance of a well-designed data model has been recognized early on in the visualization community, and there have been a number of efforts to develop a general design with a strong, formal foundation. One of the earliest was the fiber bundle model by Butler and Pendley [5]. Their model was inspired the mathematical abstraction of the same name. Fiber bundles have proven to be somewhat difficult to implement in their pure form, though the concepts have inspired several follow-on efforts. The original fiber bundle abstractions did not provide a convenient means to access

the underlying discretization (mesh) of a data set. This was a problem since many visualization algorithms operate by iterating over various types of cells of the mesh.

One system in particular that has been influenced by fiber bundle concepts is OpenDX (formerly IBM Data Explorer[13, 1]). Beginning with Haber et al [8], the fiber bundle model was adapted into a model that would support a general-purpose data-flow visualization system. OpenDX can handle fields with node association indices 0 or B, where B is the base dimensionality of the field. OpenDX does not support adaptive meshes, though more recent work by Treinish [23] describes a model that would accommodate such data.

Another field modeling effort was the Field Encapsulation Library (FEL) project, first presented at Visualization '96 [3]. FEL excelled with the multi-block curvilinear grids that are popular in computational fluid dynamics applications. FEL differed from most other modeling efforts in that it defined separate class hierarchies for meshes and fields, rather than a single combined object type. A second version of FEL, FEL2, followed after a basic redesign and total rewrite [16, 15]. FEL2 introduced fundamental design features that enabled the library to operate with far larger data sets, including a consistent demand-driven evaluation model [14] and the integration of demand-paging techniques [6]. FEL2, like the original version of FEL, assumed that all objects were in $\mathbf{R}^3$ physical space, and that all fields effectively had a node association index of 0.

The Visualization Toolkit (vtk) [20], like OpenDX, is an open source visualization system with a fairly general data model. The vtk data model uses an extended concept of cells, including such primitives as polylines and triangle strips as cell types. Recent extensions [12] have focused on enabling the data model (and thus the whole system) to handle large data. Like *FM*, vtk utilizes a demand-driven evaluation strategy. In vtk, visualization techniques negotiate with a data source in order to determine appropriate streaming parameters, then the streaming commences. *FM* demand-driven evaluation is maximally fine-grained: visualization techniques request data one cell at a time, and the lazy evaluation happens at the same granularity. The *FM* approach leads to more function calls between the data consumer and producer, while the vtk approach implies that the data consumer has to know more about the characteristics of the data set it is accessing.

Another object-oriented data flow visualization system intended for large data visualization is SCIRun [2, 19]. One distinguishing characteristic of the SCIRun development effort was the focus on computational steering, i.e., analyzing data from a simulation and modifying simulation parameters, as the simulation is running. SCIRun also allowed for some mesh adaptation during a simulation run. The data model was not the primary focus of the overall development effort.

VisAD [10, 9] is a relatively general, object-oriented model for numerical data. The user can construct data objects with a style similar to expressing mathematical functions. In contrast to the models described previously, VisAD is implemented in Java. The VisAD model is quite flexible, though the Java implementation makes it less suitable for very large data. The VisAD model does put more effort into the inclusion of metadata – data about data – than most other designs. For example, VisAD provides for the specification of the units of measurement. Thus, for example, VisAD users should be less likely to confuse distances measured in miles with distances measured in kilometers.

4 Design and Implementation

Object-oriented design is hard. As Gamma et al. point out:

> Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time [7].

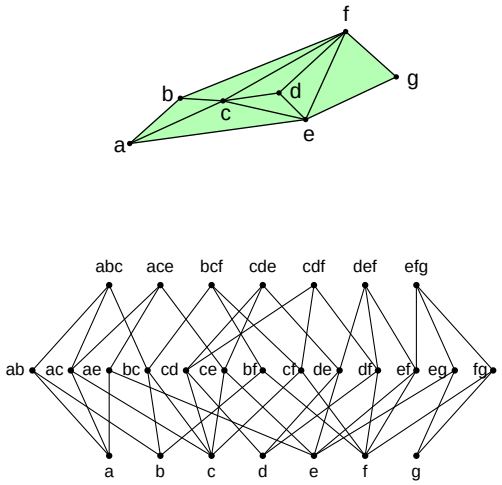


Figure 2: A small 2-mesh and its corresponding incidence graph. Answering `faces` queries is equivalent to following paths upwards or downwards in the graph.

In the case of the design of *FM*, we benefit from our experience with the original [3] and second generation [16, 15] Field Encapsulation Library (FEL) projects. Both generations had relatively demanding performance requirements from applications such as Virtual Windtunnel [4]. Both also faced large data challenges. The second generation FEL was used by several different applications, providing reuse cases that helped us refine the class interfaces.

In *FM*, as in FEL, the two main types of objects in the model are meshes and fields. We discuss key features of both types next.

4.1 Shared Mesh and Field Interface

Both mesh and field classes inherit interface from the class `FM_field_interface<B,D,T>`, where the template arguments `B`, `D` and `T` specify the base dimensionality, physical dimensionality and node type, respectively. For meshes, the node type is the coordinate type: `FM_vector<D,FM_coord>`. The interface class specifies the member functions `at_cell`, `at_base`, and `at_phys`. The `at_cell` call takes a cell argument and appends values to a C++ standard library vector [11] passed in by pointer. The `at_base` and `at_phys` member functions provide access to field values at a single point in base space or physical space, respectively. We provide detail on the access function semantics below.

4.2 Mesh Interface

In general an application can access two types of information from a mesh object: geometric and topological. Geometric information is accessed primarily through the `at_cell` call, which produces the coordinates of the vertices of its cell argument. The `at_base` call takes a point in base coordinates and produces physical coordinates, thus it provides a means to convert between the two coordinate systems. (There is also a routine to do the opposite conversion). The `at_phys` call may at first seem redundant for meshes, but via its integer return value it does provide a means for verifying whether a given physical point is within the region where the field is defined.

FM mesh objects have several member functions that provide topological information. Here we focus on one particular method, `faces`, that is key to the general node association design. To illustrate the `faces` method, we consider the small 2-mesh in Figure 2. Below the the mesh is an incidence graph which captures all the face relationships of the mesh. Each row of nodes[2] in the graph corresponds to a particular cell dimensionality, with the rows ordered by increasing dimensionality from bottom to top. The graph contains an edge between nodes representing a k-cell σ and a $(k+1)$-cell τ if σ is a face of τ. The `faces` methods takes a k-cell σ and an integer argument j. If $j < k$, then `faces` returns the j-cells that are faces of σ. If $j > k$, then `faces` returns the j-cells that σ is the face of. If $j = k$, then `faces` returns σ. In terms of the graph in Figure 2, the $j < k$ case is equivalent to following all paths downward to the jth row from the node corresponding to σ; the $j > k$ case is equivalent to following all paths upward instead of downward. For those familiar with algebraic topology, the functionality of the `faces` call is essentially equivalent to the closure and star operators combined [17]. The `faces` method has many uses, for example it may be used for obtaining the edges of a given hexahedron. We will see how `faces` is used in conjunction with general node association below in Section 4.4.

The *FM* mesh interface also supports iterator functionality compatible with the C++ standard library [11]. Meshes behave as collections of cells, and one can iterate over the cells. Unlike standard library collections, mesh objects provide a richer set of iteration possibilities. Typically one wants to iterate over cells of a particular dimension, or some other subset of the total collection of cells. *FM* provides this control via optional arguments to the `begin` iterator initializer call. Other than that difference, *FM* iterator style is compatible with the standard library, and one should be able use any of the standard library algorithms that operate with a collection that provides a `const_iterator`.

4.3 Mesh Implementation

Figure 3 summarizes the *FM* mesh hierarchy. All mesh objects share common interface defined by `FM_mesh_` and `FM_mesh<B,D>`. The subclasses also share implementation through inheritance. For example, topological methods such as `faces` are implemented in `FM_structured_mesh<B,D>` and used by all structured mesh subclasses. Meshes are also responsible for point location and contribute geometric information that is used for interpolation. Efficient point location is critical in a high-performance field model, as it is an intermediate step when computing field values at arbitrary points in space. Through the class hierarchy we are able to provide point location routines that exploit characteristics of various types of meshes in order to provide increased performance.

4.4 Field Interface

Fields are all templated on base dimensionality, physical dimensionality and node type. *FM* uses the same source for scalar, vector and in general tensor fields — all are instantiated from the same class definitions. The fundamental field member function for obtaining field values is `at_cell`, which produces one or more field values, returning them in a C++ standard library vector object [11]. For a field with node association index k, an `at_cell` call with a k-cell argument will produce a single field value. The same call with a j-cell argument, $j \neq k$, first would use the `faces` call on the underlying mesh to convert the j-cell into a collection of k-cells. Then, for each of the resulting k-cells the field would append a single value to the result collection. Thus, for example, a node association index 0 field given a hexahedron argument would produce

[2]Note that *graph nodes* and *field nodes* are different concepts.

```
FM_mesh_
  FM_mesh<B,D>
    FM_structured_mesh<B,D>
      FM_curvilinear_mesh<B,D>
      FM_regular_interval
      FM_irregular_interval
      FM_product_mesh<B,D>
        FM_regular_mesh<B,D>
    FM_unstructured_mesh<B,D>
    FM_multi_mesh<B,D>
    FM_time_series_mesh<B,D>
```

Figure 3: A synopsis of the main mesh classes, with the inheritance hierarchy signified by indentation. The integer template arguments `B` and `D` to `FM_mesh<B,D>` and the classes below it specify the base and physical dimensionality, respectively. The `FM_mesh_` parent class provides a convenient handle when an application only requires the portion of the mesh interface that is not dependent upon the mesh dimensionality, e.g., the iterator interface.

```
FM_field_
  FM_field<B,D,T>
    FM_core_field<B,D,T>
    FM_multi_field<B,D,T>
    FM_unary_derived_field<B,D,T>
    FM_binary_derived_field<B,D,T>
    FM_time_series_field<B,D,T>
```

Figure 4: A synopsis of the main field classes, with the inheritance hierarchy signified by indentation. The `B` and `D` template arguments have the same meaning as for meshes. The `T` template argument specifies the field node type, e.g., `float`. The purpose of the `FM_field_` parent class is analogous to that of `FM_mesh_`: it provides a convenient handle when an application only requires the portion of the field interface that is not dependent upon the template arguments.

8 values, 1 for each vertex. Or, for example, a node association index 3 ("finite volume") field `at_cell` call with a vertex argument would return in general 8 values. We say "in general" since a vertex at the boundary of the mesh is the face of fewer than 8 hexahedra.

The utility of the `at_cell` definition becomes further apparent when we consider cases where we have a field with one particular node association index, but want it to behave like another. Our approach would be to define an adapter class [7], derived from `FM_field<B,D,T>`, with its own `at_cell` method implementation. For instance, consider the case where we have a visualization algorithm that expects a single value when calling `at_cell` with a vertex, but our field has a node association index not equal to 0. The adapter would take an incoming vertex argument and call `at_cell` on the adapted field. The multiple values received in response could be averaged (perhaps with some weighting factors) to produce the final single value response. Such an adapter would enable us to reuse some older visualization techniques that make vertex-centered data assumptions.

4.5 Field Implementation

The `FM_field_` class hierarchy is summarized in Figure 4. The subclasses are primarily responsible for providing implementations for the `at_cell` member function. Core fields produce values from a memory buffer. `FM_multi_field<B,D,T>` represents fields consisting of multiple subfields; `at_cell` calls are delegated to the appropriate subfield. The derived field classes produce values on demand, applying a mapping function to the values produced by

Mesh	$Card_0$	$Card_1$	$Card_2$
Initial	14605	43009	28404
Level 1	26189	88991	59000
Level 2	62926	235331	156498
Level 3	169933	662344	441147
Level 4	380877	1505024	1003313
Level 5	488574	1935619	1291834

Table 1: Cell set cardinalities for the 0-cells, 1-cells, and 2-cells in the airfoil example illustrated in Figure 5. The levels denote the number of refinement steps taken.

`at_cell` calls on the underlying fields.

The `FM_field<B,D,T>` class provides default implementations for the `at_phys` and `at_base` methods. Both implementations operate by locating a cell containing the given point, obtaining field values in the neighborhood of the point using `at_cell`, and then interpolating based on the geometry of the cell. Since both `at_phys` and `at_base` are implemented in terms of `at_cell`, field subclasses are not required to provide implementations of these two functions. Nevertheless, some subclasses do provide their own implementations in order to employ optimizations that are specific to certain field types.

4.6 Time

In the previous sections we have said little about time, but this is not because time-varying data is unimportant. To the contrary, many large data sets come in the form of a time series. There are two main strategies we could choose in order to address the needs of time-varying data in *FM*. One approach would be to simply treat time as an added dimension, utilizing the general-dimension mechanisms in we have already developed. The alternative would be to treat time as special, distinct from the spatial coordinates. At first glance, the former strategy may seem more appealing – we would like to reuse implementation when we can – but we decided to go the latter route instead, for several reasons. First, the spatial and temporal resolutions of the data can be dramatically different. Especially in post-processing applications, what is saved of the simulation is typically down sampled in time from the resolution used during the run. This implies we may want to do spatial and temporal interpolation differently. Second, many visualization techniques are designed to work at some instance in time, and they do not handle time explicitly. If time were an added dimension, then the user of *FM* would need to reduce the data dimensionality before passing the data to the visualization technique. While such a design is possible, we concluded that it could be somewhat awkward for our users, and there is some question as to how great of a performance hit we would take if we were to employ such an approach.

In *FM*, time-varying meshes and fields are represented by the classes `FM_time_series_mesh<B,D>` and `FM_time_series_field<B,D,T>`, respectively. Base position, physical position, and cell arguments all contain a time member. Objects that do not vary with time ignore this member. Time series objects use the time member to index into the series and as part of the interpolation process when needed. Within this design, visualization techniques are free to request values in space and time as needed by the particular algorithms.

5 Results

5.1 2-D TAG

As a demonstration of the *Field Model* capabilities, we consider a 2-D Triangular Adaptive Grid (TAG) code that has served as the basis for previous research efforts on adaptive grid techniques [18]. The TAG system is designed to be relatively insulated from a particular flow solver. TAG provides mesh geometry and connectivity information used by the solver; the solver in turn computes field node values and adaptation criteria that are associated with the mesh edges. Based on the adaptation criteria, the TAG system refines or coarsens the mesh. Figure 5 illustrates the airfoil test case that we consider here. Table 1 quantifies the mesh size in terms of the number of k-cells, $k = 0..2$, for each level of refinement.

Our motivation for choosing the TAG 2-D example is to test extensibility, in particular, with an adaptive mesh object. It is neither feasible nor desirable for *FM* to provide built-in support for every mesh data structure; the library implementation would become too bulky and difficult to maintain. Instead, our goal is a design that is modular enough that new types of meshes can be added without significant modification to existing parts of the model. To be successful in this endeavor, we have three criteria. First, the class interfaces should be general enough to be applicable to a variety of object types. So far we consider ourselves to have met this requirement. *FM* can represent a variety of objects, including structured and unstructured objects and multi-block objects. We have not encountered significant limitations due to the interfaces. Second, the interface abstractions should not cause us to suffer an unacceptable loss in performance. We address this issue below. Finally, the class design should support reuse of parts of the implementation, so that newly introduced mesh and field types do not have to reimplement common routines. Our design has been successful in this respect as well. For 2-D TAG, we defined a new class `TAG2D_unstructured_mesh`, which is derived from `FM_unstructured_mesh<2,2>`. Note that the TAG2D class is not templated; the base dimensionality and physical dimensionality are hard-coded in the 2-D TAG implementation that we obtained. Our TAG2D class must provide implementation of some basic member functions such as `at_cell` and `faces`, since these functions refer to TAG-specific data structures. Other functionality, such as iterator support, is inherited from `FM_unstructured_mesh<2,2>`; our TAG class can reuse the existing code.

The version of the 2-D TAG code we adapted for our example here executes serially. Oliker and Biswas [18] also have versions of the same code designed for parallel architectures, including message passing systems. We do not have experience yet with how well *FM* would accommodate such generalizations, but we are interested in investigating this. There is also a 3-D version of the adaptive grid code, developed by the same research group, that is analogous in many respects to 2-D TAG. The 3-D version contains non-simplicial cells, including pyramids, prisms and hexahedra, which should provide some additional challenge, although we do not anticipate any fundamental problems adapting such objects.

5.2 Performance

Field Model at its heart is about abstractions, and it is natural to ask what cost one has to pay for the benefits of abstraction. This in general is a difficult question to answer, because:

- cost is relative to some alternative, and what alternatives we have vary from case to case;
- how much abstraction overhead is apparent depends on the balance between data access and computation using the data;
- with large data, access time can be significantly influenced by the locality or lack thereof in data access patterns.

Despite these difficulties, it is still important to quantify the performance of the data model. We present the results from two initial

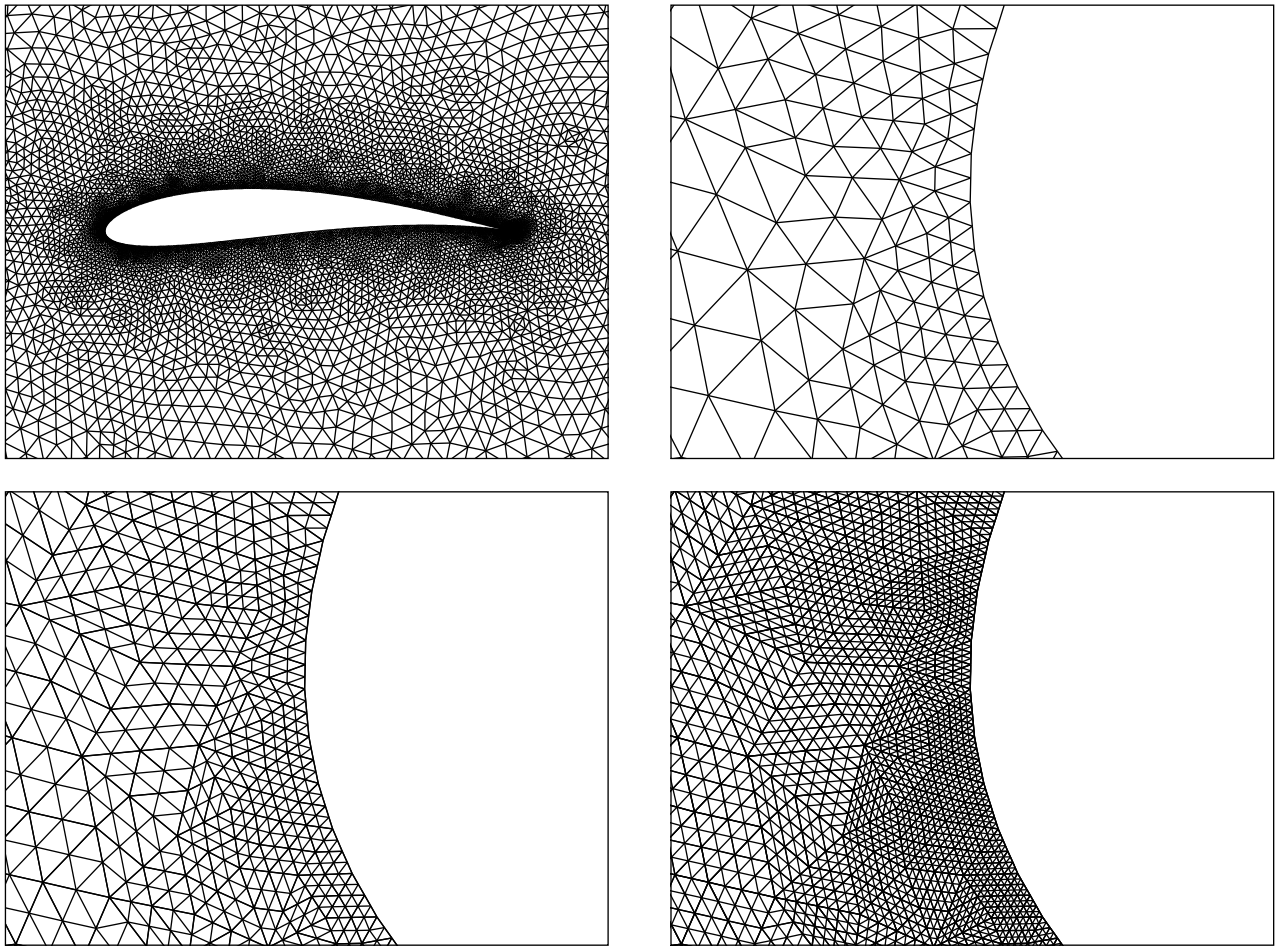


Figure 5: The airfoil data set with 2-D TAG. At the upper left is a close-up of the whole airfoil. At the upper right is a much closer view of the leading edge of the airfoil. The two images in the lower row display successive refinement iterations within the same region.

Mesh	Bounding Box		Edge Drawing	
	Abstract	Direct	Abstract	Direct
Initial	13.9	1.1	1395.9	1324.5
Level 1	23.5	1.8	2634.7	2555.9
Level 2	56.3	4.3	6929.4	6583.7
Level 3	152.4	11.7	20276.7	18841.9
Level 4	343.8	26.4	45754.7	44351.6
Level 5	462.8	35.7	59516.6	58007.2

Table 2: Initial *FM* example timings, in msec. The "Bounding Box" columns illustrate a worst case scenario for *FM*: we compare an algorithm written using *FM* to a hand-coded implementation that accesses the data structures directly, and the amount of compute relative to each data access is small. In this scenario the *FM* version comes out over an order of magnitude slower. The "Edge Drawing" columns illustrate a scenario that may be more typical. Once again we compare an algorithm written using *FM* to a hand-coded implementation that accesses the data structures directly, but in our second scenario the compute time is more significant. In this second scenario, the *FM* version is slower, but by roughly only 5%.

tests based on the 2-D TAG example discussed in the previous section. Our first test involves computing the bounding box of the TAG mesh. This test is in many respects a worst case scenario because we compare the abstract *FM* method to a hand-coded C-style implementation that has direct access to the data buffers, the amount of computation using the data is minimal, and the data are not really large enough for cache-miss rates to dominate. The columns under "Bounding Box" in Table 2 summarize the results for the example airfoil data set, measured on a 195 MHz, dual processor SGI Onyx2 workstation with 512M of memory. The worst case does look pretty bad: the difference in total times in each case is over an order of magnitude. Still, depending on the application, the abstract performance may be fast enough.

As a second example, we consider a scenario where we generate postscript images consisting of the edges in the TAG mesh. We time the actual code we used to generate the images in Figure 5. Like the first scenario, we compare access through *FM* to hand-coded direct access to the data structures. Unlike the first scenario, the computation involves some simple transformations followed by a write to our postscript file. This is clearly more expensive than our bounding box computation. The columns under "Edge Drawing" summarize the results. The *FM* version runs slower, but by roughly only 5%. For this application the overhead is likely to be acceptable.

The timings in Table 2 clearly are not a thorough assessment of *FM* performance. *Field Model* is still relatively early in its development process, and we have done little performance tuning so far. Our plan is to port the VisTech library [21] to *FM* in the near future. VisTech consists of a collection of standard visualization algorithms, written in terms of FEL2 [16, 15]. We will be able to compare *FM*/VisTech performance with that of FEL2/VisTech, and in some cases with implementations hand-coded for specific mesh and field types. VisTech applications will provide examples with more typical balance between data access and computation as well as relatively typical data access patterns for visualization applications.

6 Conclusion

We have presented an overview of *Field Model* (*FM*), an object-oriented data model for mesh and field data. *FM* benefits significantly from our experiences with FEL2 [16, 15], an earlier effort focused on the development of high-performance library for large data. *FM* goes beyond FEL2 in generality: *FM* can represent data with general base and physical dimensionality as well as fields with general node association. Furthermore, we anticipate that *FM* will be able to successfully handle adaptive mesh types. Our experience so far with the 2-D TAG [18] adaptive code confirms our expectations.

Two of the primary design goals of the *FM* project are modularity and extensibility. Our vision is that *FM* will serve as a common model where others in the community can contribute extensions specific to their mesh and field objects. The incentive would be that data brought into the shared model could be analyzed by what we hope will be a wide collection of analysis techniques written in terms of the model. Towards this end, we are working to establish *FM* as an Open Source [22] project, with its development home on SourceForge. We have established a site there (`http://field-model.sourceforge.net`), and we currently have a few initial files uploaded to the repository. We anticipate that by Vis'01 all the source used to create objects such as those displayed in this article will be available from our site.

Acknowledgements

We would like to thank Ernst Mücke for the interlinked tori point set used in Figure 1. We would also like to thank Rupak Biswas for providing the 2-D TAG code and example data used in Section 5.1 We are also grateful to Pete Vanderbilt and all the members of the Data Analysis Group for helpful insights. Finally, we would like to thank VA Linux for their ongoing support of the Open Source [22] software movement, and SourceForge in particular.

References

[1] G. Abram and L. Treinish. An extended data-flow architecture for a data analysis and visualization. In *Proceedings of Visualization '95*, pages 263–270. IEEE Computer Society Press, 1995.

[2] S. Parker amd D. Weinstein and C. Johnson. The SCIRun computational steering software system. In E. Arge, A. Bruaset, and H. Langtangen, editors, *Modern Software Tools for Scientific Computing*. Birkhäuser, 1997.

[3] S. Bryson, D. Kenwright, and M. Gerald-Yamasaki. FEL: The field encapsulation library. In *Proceedings of Visualization '96*, pages 241–247. IEEE Computer Society Press, October 1996.

[4] S. Bryson and C. Levit. The virtual windtunnel: An environment for the exploration of three-dimensional unsteady flows. In *Proceedings of Visualization '91*. IEEE Computer Society Press, October 1991.

[5] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45–51, Sep/Oct 1989.

[6] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of Visualization '97*, pages 235–244. IEEE Computer Society Press, October 1997.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] R. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *Proceedings of Visualization '91*, pages 298–305, October 1991.

[9] W. Hibbard. VisAD: Connecting people to computations and people to people. *Computer Graphics*, 32(3), 1998.

[10] W. Hibbard, C.R. Dyer, and B. Paul. A lattice data model for data display. In *Proceedings of Visualization '94*, pages 310–317, October 1994.

[11] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

[12] C. Law, K. Martin, W. Schroeder, and J. Temkin. A multithreaded streaming pipeline architecture for large structured data sets. In *Proceedings of Visualization '99*, pages 225–232, October 1999.

[13] B. Lucas et al. An architecture for a scientific visualization system. In *Proceedings of Visualization '92*, pages 107–114. IEEE Computer Society Press, 1992.

[14] P. Moran and C. Henze. Large data visualization with demand-driven calculation. In *Proceedings of Visualization '99*, pages 27–33. IEEE Computer Society Press, October 1999.

[15] P. Moran and C. Henze. The FEL 2.2 reference manual. Technical report, National Aeronautics and Space Administration, 2000. NAS-00-007.

[16] P. Moran, C. Henze, and D. Ellsworth. The FEL 2.2 user guide. Technical report, National Aeronautics and Space Administration, 2000. NAS-00-002.

[17] J. R. Munkres. *Elements of Algebraic Topology*. Addison-Wesley, 1984.

[18] L. Oliker and R. Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Proceedings of SC '99*. IEEE Computer Society Press, November 1999.

[19] S. Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, University of Utah, 1999.

[20] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice-Hall Inc., New Jersey, second edition, 1997.

[21] H.-W. Shen, T. Sandstrom, D. Kenwright, and L.-J. Chiang. *VisTech Library User and Programmer Guide*. National Aeronautics and Space Administration, 1999.

[22] Open Source. `http://www.opensource.org`.

[23] L. A. Treinish. A function-based data model for visualization. In *Visualization '99 Late Breaking Hot Topics*. IEEE Computer Society Press, 1999.

Appendix A

We provide the *Field Model* (*FM*) source for the examples presented in the body of this report in the following pages.

```
// Emacs mode -*-c++-*- //
#ifndef _FM_BASE_H_
#define _FM_BASE_H_
/*
 * NAME: FM_base.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_vector.h"
#include "FM_submesh_id.h"
#include "FM_time.h"


template <int B, typename T = FM_coord>
class FM_base : public FM_vector<B,T>
{
public:
  FM_time<T> time;
  FM_submesh_id submesh_id;

  FM_base() {}
  FM_base(const FM_time<T>& t, const FM_submesh_id& sid) :
    time(t), submesh_id(sid) {}
  FM_base(const FM_vector<B,T>& v) : FM_vector<B,T>(v) {}
  FM_base(const FM_vector<B,T>& v,
          const FM_time<FM_u32>& t, const FM_submesh_id& sid) :
    FM_vector<B,T>(v), time(t), submesh_id(sid) {}
};

template <typename T>
class FM_base<1,T> : public FM_vector<1,T>
{
public:
  FM_time<T> time;
  FM_submesh_id submesh_id;

  FM_base() {}
  FM_base(const FM_time<T>& t, const FM_submesh_id& sid) :
    time(t), submesh_id(sid) {}
  FM_base(const FM_vector<1,T>& v) : FM_vector<1,T>(v) {}
  FM_base(T c) : FM_vector<1,T>(c) {}
  FM_base(T c, const FM_time<FM_u32>& t, const FM_submesh_id& sid) :
    FM_vector<1,T>(c), time(t), submesh_id(sid) {}
};

template <typename T>
class FM_base<2,T> : public FM_vector<2,T>
{
public:
  FM_time<T> time;
  FM_submesh_id submesh_id;

  FM_base() {}
  FM_base(const FM_time<T>& t, const FM_submesh_id& sid) :
    time(t), submesh_id(sid) {}
  FM_base(const FM_vector<2,T>& v) : FM_vector<2,T>(v) {}
  FM_base(T c0, T c1) : FM_vector<2,T>(c0, c1) {}
  FM_base(T c0, T c1, const FM_time<T>& t, const FM_submesh_id& sid) :
    FM_vector<2,T>(c0, c1), time(t), submesh_id(sid) {}
};

template <typename T>
class FM_base<3,T> : public FM_vector<3,T>
{
public:
  FM_time<T> time;
  FM_submesh_id submesh_id;

  FM_base() {}
  FM_base(const FM_time<T>& t, const FM_submesh_id& sid) :
    time(t), submesh_id(sid) {}
  FM_base(const FM_vector<3,T>& v) : FM_vector<3,T>(v) {}
  FM_base(T c0, T c1, T c2) : FM_vector<3,T>(c0, c1, c2) {}
  FM_base(T c0, T c1, T c2,
          const FM_time<FM_u32>& t, const FM_submesh_id& sid) :
    FM_vector<3,T>(c0, c1, c2), time(t), submesh_id(sid) {}
};


template <int B, typename T>
std::ostream& operator<<(std::ostream& lhs, const FM_base<B,T>& rhs)
{
  lhs << "(";
  int i;
  for (i = 0; i < B; i++) {
    if (i > 0) lhs << ", ";
    lhs << rhs[i];
  }
  if (rhs.time.defined()) {
    if (i++ > 0) lhs << ", ";
    lhs << "time=" << rhs.time;
  }
  if (rhs.submesh_id.defined()) {
    if (i++ > 0) lhs << ", ";
    lhs << "submesh_id=" << rhs.submesh_id;
  }
  return lhs << ")";
}



/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_CELL_H_
#define _FM_CELL_H_
/*
 * NAME: FM_cell.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <stdexcept>
#include <iostream>
#include <new>
#include <typeinfo>
#include <assert.h>
#include "FM_shared_object.h"
#include "FM_submesh_id.h"
#include "FM_vector.h"
#include "FM_base.h"
#include "FM_time.h"
#include "FM_combinatorics.h"
#include "FM_ostringstream.h"

enum FM_cell_type_enum
{
  FM_OTHER_CELL = -1,
  FM_VERTEX_CELL = 0,
  FM_EDGE_CELL = 1,
  FM_TRIANGLE_CELL = 2,
  FM_TETRAHEDRON_CELL = 3,
  FM_QUADRILATERAL_CELL = 4,
  FM_PYRAMID_CELL = 5,
  FM_PRISM_CELL = 6,
  FM_HEXAHEDRON_CELL = 8
};

FM_u32 FM_cell_type_to_dimension(FM_cell_type_enum ct)
{
  const FM_u32 dimension[9] = {0, 1, 2, 3, 2, 3, 3, 0, 3};
  if (ct == FM_OTHER_CELL) {
    FM_ostringstream err;
    err << "FM_cell_type_to_dimension(FM_OTHER_CELL) undefined";
    throw std::logic_error(err.str());
  }
  return dimension[ct];
}

class FM_mesh_;

class FM_cell : public FM_shared_object
{
public:
  FM_cell() {}
  FM_cell(const FM_time<FM_u32>& t, const FM_submesh_id& s) :
    time(t), submesh_id(s) {}
  FM_cell(const FM_cell& c) : time(c.time), submesh_id(c.submesh_id) {}

  virtual ~FM_cell() {}

  FM_time<FM_u32> get_time() const { return time; }

  virtual void set_time(const FM_time<FM_u32>& t) { time = t; }

  FM_submesh_id get_submesh_id() const { return submesh_id; }

  virtual void set_submesh_id(const FM_submesh_id& sid) { submesh_id = sid; }

  virtual FM_u32 get_dimension() const = 0;

  virtual FM_u32 get_n_faces(FM_u32) const = 0;

  virtual FM_cell_type_enum get_type() const = 0;

  virtual bool is_equal_to(const FM_cell&) const = 0;

  virtual bool is_subsimplex() const { return false; }

  virtual std::ostream& str(std::ostream& o) const = 0;

  virtual void
  structured_mesh_vertex_indices(const FM_mesh_*, FM_u32*, FM_u64[]) const
  {
    FM_ostringstream err;
    err << *this << "::structured_mesh_vertex_indices(): not allowed";
    throw std::logic_error(err.str());
  }

protected:
  FM_time<FM_u32> time;
  FM_submesh_id submesh_id;
};


bool operator==(const FM_cell& lhs, const FM_cell& rhs)
{
  return lhs.is_equal_to(rhs);
}

bool operator!=(const FM_cell& lhs, const FM_cell& rhs)
{
  return !(lhs == rhs);
}


template <int B> class FM_structured_mesh_0_cell_iter_impl;
template <int B> class FM_structured_mesh_B_cell_iter_impl;


template <int B>
class FM_structured_cell : public FM_cell
{
protected:
  FM_structured_cell() {}

  FM_structured_cell(const FM_vector<B,FM_u32>& i) :  indices(i) {}

  FM_structured_cell(const FM_time<FM_u32>& t, const FM_submesh_id& sid,
                     const FM_vector<B,FM_u32>& i) :
    FM_cell(t, sid), indices(i) {}

  friend class FM_structured_mesh_0_cell_iter_impl<B>;
  friend class FM_structured_mesh_B_cell_iter_impl<B>;

public:
  virtual FM_u32 get_alignment() const
  {
    FM_ostringstream err;
    err << *this << "::get_alignment(): not allowed";
    throw std::logic_error(err.str());
  }

  virtual void set_alignment(FM_u32)
  {
    FM_ostringstream err;
    err << *this << "::set_alignment(): not allowed";
    throw std::logic_error(err.str());
  }

  virtual FM_u32 get_subid() const
  {
    FM_ostringstream err;
    err << *this << "::get_subid(): not allowed";
    throw std::logic_error(err.str());
  }

  virtual void set_subid(FM_u32)
  {
    FM_ostringstream err;
    err << *this << "::set_subid(): not allowed";
    throw std::logic_error(err.str());
  }

  const FM_u32* get_indices() const { return indices.v(); }

  FM_u32 get_index(FM_u32 k) const { return indices[k]; }

  virtual void set_indices(const FM_u32 i[])
  {
    for (int j = 0; j < B; j++)
      indices[j] = i[j];
  }

  virtual void set_index(FM_u32 k, FM_u32 i) { indices[k] = i; }

  virtual FM_cell_type_enum get_type() const
  {
    FM_cell_type_enum res;
    switch(get_dimension()) {
    case 0:
      res = FM_VERTEX_CELL;
      break;
    case 1:
      res = FM_EDGE_CELL;
      break;
    case 2:
      res = is_subsimplex() ? FM_TRIANGLE_CELL : FM_QUADRILATERAL_CELL;
      break;
    case 3:
      res = is_subsimplex() ? FM_TETRAHEDRON_CELL : FM_HEXAHEDRON_CELL;
      break;
    default:
      res = FM_OTHER_CELL;
    }
    return res;
  }

  virtual FM_u32 get_n_faces(FM_u32 k) const
  {
    FM_u32 d = get_dimension();
    return FM_choose(d, (d - k)) * FM_pow_2(d - k);
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    switch(get_dimension()) {
    case 0:
      o << "V";
      break;
    case 1:
      o << "E";
      break;
    case 2:
      o << (is_subsimplex() ? "F" : "Q");
      break;
    case 3:
      o << (is_subsimplex() ? "T" : "H");
      break;
    default:
      o << "H" << get_dimension();  // d-cube
    }

    o << "(";
    FM_u32 j;
    for (j = 0; j < B; j++) {
      if (j > 0) o <<  ", ";
      o << indices[j];
    }
    if (is_subsimplex()) {
      if (j++ > 0) o <<  ", ";
      o << "subid=" << get_subid();
    }
```

```
    else {
      if (!(get_dimension() == 0 || get_dimension() == B)) {
        if (j++ > 0) o <<  ", ";
        o << "alignment=" << get_alignment();
      }
    }
    if (time.defined()) {
      if (j++ > 0) o <<  ", ";
      o << "time=" << time;
    }
    if (submesh_id.defined()) {
      if (j++ > 0) o <<  ", ";
      o << "submesh_id=" << submesh_id;
    }
    return o << ")";
  }

protected:
  FM_vector<B,FM_u32> indices;
};


template <int B> class FM_structured_0_cell;
template <int B> class FM_structured_B_cell;


template <int B>
class FM_structured_k_cell : public FM_structured_cell<B>
{
public:
  FM_structured_k_cell(FM_u32 k, FM_u32 a, const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(i), dimension(k), alignment(a) {}

  FM_structured_k_cell(const FM_time<FM_u32>& t, const FM_submesh_id& sid,
                       FM_u32 k, FM_u32 a, const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(t, sid, i), dimension(k), alignment(a) {}

  virtual FM_u32 get_dimension() const { return dimension; }

  virtual void set_dimension(FM_u32 d) { dimension = d; }

  virtual FM_u32 get_alignment() const { return alignment; }

  virtual void set_alignment(FM_u32 a) { alignment = a; }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_structured_k_cell<B>* skc =
      dynamic_cast<const FM_structured_k_cell<B>*>(&c);
    if (skc) {
      return
        time == skc->time &&
        submesh_id == skc->submesh_id &&
        dimension == skc->dimension &&
        alignment == skc->alignment &&
        indices == skc->indices;
    }
    if (dimension == 0) {
      const FM_structured_0_cell<B>* s0c =
        dynamic_cast<const FM_structured_0_cell<B>*>(&c);
      if (s0c) {
        return
          time == s0c->get_time() &&
          submesh_id == s0c->get_submesh_id() &&
          indices == s0c->get_indices();
      }
    }
    else if (dimension == B) {
      const FM_structured_B_cell<B>* sBc =
        dynamic_cast<const FM_structured_B_cell<B>*>(&c);
      if (sBc) {
        return
          time == sBc->get_time() &&
          submesh_id == sBc->get_submesh_id() &&
          indices == sBc->get_indices();
      }
    }
    return false;
  }

  virtual void
  structured_mesh_vertex_indices(const FM_mesh_*, FM_u32*, FM_u64[]) const;

private:
  FM_u32 dimension;
  FM_u32 alignment;
};


template <int B>
class FM_structured_0_cell : public FM_structured_cell<B>
{
public:
  FM_structured_0_cell(const FM_time<FM_u32>& t, const FM_submesh_id& sid,
                       const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(t, sid, i) {}

  FM_structured_0_cell(const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(i) {}

  FM_structured_0_cell() {}

  FM_structured_0_cell(const FM_base<B,FM_u32>& b) :
    FM_structured_cell<B>(b.time, b.submesh_id, b) {}

  virtual FM_u32 get_dimension() const { return 0; }

  virtual FM_u32 get_n_faces(FM_u32 d) const { return d == 0 ? 1 : 0; }

  virtual FM_cell_type_enum get_type() const { return FM_VERTEX_CELL; }

  virtual FM_u32 get_alignment() const { return 0; }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_structured_0_cell<B>* s0c =
      dynamic_cast<const FM_structured_0_cell<B>*>(&c);
    if (s0c) {
      return
        time == s0c->time &&
        submesh_id == s0c->submesh_id &&
        indices == s0c->indices;
    }
    const FM_structured_k_cell<B>* skc =
      dynamic_cast<const FM_structured_k_cell<B>*>(&c);
    if (skc) {
      return
        time == skc->get_time() &&
        submesh_id == skc->get_submesh_id() &&
        0 == skc->get_dimension() &&
        indices == skc->get_indices();
    }
    return false;
  }

  virtual void
  structured_mesh_vertex_indices(const FM_mesh_*, FM_u32*, FM_u64[]) const;
};


template <int B>
class FM_structured_B_cell : public FM_structured_cell<B>
{
public:
  FM_structured_B_cell(const FM_time<FM_u32>& t, const FM_submesh_id& sid,
                       const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(t, sid, i) {}

  FM_structured_B_cell(const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(i) {}

  FM_structured_B_cell() {}

  virtual FM_u32 get_dimension() const { return FM_u32(B); }

  virtual FM_u32 get_alignment() const { return 0; }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_structured_B_cell<B>* sBc =
      dynamic_cast<const FM_structured_B_cell<B>*>(&c);
    if (sBc) {
      return
        time == sBc->time &&
        submesh_id == sBc->submesh_id &&
        indices == sBc->indices;
    }
    const FM_structured_k_cell<B>* skc =
      dynamic_cast<const FM_structured_k_cell<B>*>(&c);
    if (skc) {
      return
        time == skc->get_time() &&
        submesh_id == skc->get_submesh_id() &&
        FM_u32(B) == skc->get_dimension() &&
        indices == skc->get_indices();
    }
    return false;
  }

  virtual void
  structured_mesh_vertex_indices(const FM_mesh_*, FM_u32*, FM_u64[]) const;
};


template <int B>
class FM_structured_subsimplex : public FM_structured_cell<B>
{
public:
  FM_structured_subsimplex(FM_u32 d, FM_u32 s,
                           const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(i), dimension(d), subid(s) {}

  FM_structured_subsimplex(const FM_time<FM_u32>& t,
                           const FM_submesh_id& sid,
                           FM_u32 d, FM_u32 s,
                           const FM_vector<B,FM_u32>& i) :
    FM_structured_cell<B>(t, sid, i), dimension(d), subid(s) {}

  virtual FM_u32 get_dimension() const { return dimension; }

  virtual void set_dimension(FM_u32 d) { dimension = d; }

  virtual FM_u32 get_subid() const { return subid; }

  virtual void set_subid(FM_u32 s) { subid = s; }

  virtual FM_u32 get_n_faces(FM_u32 k) const
  {
    if (k > dimension)
      return 0;
    return FM_choose(dimension + 1, k + 1);
  }

  virtual bool is_subsimplex() const { return true; }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_structured_subsimplex<B>* ss =
      dynamic_cast<const FM_structured_subsimplex<B>*>(&c);
    if (ss) {
```

```
      return
        time == ss->time &&
        submesh_id == ss->submesh_id &&
        dimension == ss->dimension &&
        subid == ss->subid &&
        indices == ss->indices;
    }
    return false;
  }

  virtual void
  structured_mesh_vertex_indices(const FM_mesh_*, FM_u32*, FM_u64[]) const;

private:
  FM_u32 dimension;
  FM_u32 subid;
};


class FM_unstructured_cell : public FM_cell
{
public:
  const FM_u32 dimension;
  const FM_u64 index;

  FM_unstructured_cell(const FM_time<FM_u32>& t, const FM_submesh_id& s,
                       FM_u32 d, FM_u64 i) :
    FM_cell(t, s), dimension(d), index(i) {}

  FM_unstructured_cell(const FM_unstructured_cell& uc, FM_u32 d, FM_u64 i) :
    FM_cell(uc), dimension(d), index(i) {}

  virtual FM_u32 get_dimension() const { return dimension; }
};


class FM_unstructured_simplex : public FM_unstructured_cell
{
public:
  FM_unstructured_simplex(const FM_time<FM_u32>& t, const FM_submesh_id& s,
                          FM_u32 d, FM_u64 i) :
    FM_unstructured_cell(t, s, d, i) {}

  FM_unstructured_simplex(const FM_unstructured_simplex& us,
                          FM_u32 d, FM_u64 i) :
    FM_unstructured_cell(us, d, i) {}

  virtual FM_cell_type_enum get_type() const
  {
    const FM_cell_type_enum types[4] = {
      FM_VERTEX_CELL, FM_EDGE_CELL, FM_TRIANGLE_CELL, FM_TETRAHEDRON_CELL
    };
    return dimension <= 3 ? types[dimension] : FM_OTHER_CELL;
  }

  virtual FM_u32 get_n_faces(FM_u32 k) const
  {
    return FM_choose(dimension + 1, k + 1);
  }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_unstructured_simplex* us =
      dynamic_cast<const FM_unstructured_simplex*>(&c);
    if (us == 0) return false;
    return
      time == us->time &&
      submesh_id == us->submesh_id &&
      dimension == us->dimension &&
      index == us->index;
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    const char letters[] = "VEFT";
    if (dimension < 4)
      o << letters[dimension];
    else
      o << "S" << dimension;
    o << "(" << index;
    if (time.defined())
      o <<  ", time=" << time;
    if (submesh_id.defined())
      o <<  ", submesh_id=" << submesh_id;
    return o << ")";
  }
};


class FM_unstructured_pyramid : public FM_unstructured_cell
{
public:
  FM_unstructured_pyramid(const FM_time<FM_u32>& t, const FM_submesh_id& s,
                          FM_u64 i) :
    FM_unstructured_cell(t, s, 3, i) {}

  FM_unstructured_pyramid(const FM_unstructured_cell& uc, FM_u64 i) :
    FM_unstructured_cell(uc, 3, i) {}

  virtual FM_cell_type_enum get_type() const { return FM_PYRAMID_CELL; }

  virtual FM_u32 get_n_faces(FM_u32 k) const
  {
    const FM_u32 n_faces[] = {4, 8, 5, 1};
    return k <= 3 ? n_faces[k] : 0;
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    o << "P(" << index;
    if (time.defined())
      o <<  ", time=" << time;
    if (submesh_id.defined())
      o <<  ", submesh_id=" << submesh_id;
    return o << ")";
  }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_unstructured_pyramid* up =
      dynamic_cast<const FM_unstructured_pyramid*>(&c);
    if (up == 0) return false;
    return
      time == up->time &&
      submesh_id == up->submesh_id &&
      index == up->index;
  }
};


class FM_unstructured_prism : public FM_unstructured_cell
{
public:
  FM_unstructured_prism(const FM_time<FM_u32>& t, const FM_submesh_id& s,
                        FM_u64 i) :
    FM_unstructured_cell(t, s, 3, i) {}

  FM_unstructured_prism(const FM_unstructured_cell& uc, FM_u64 i) :
    FM_unstructured_cell(uc, 3, i) {}

  virtual FM_cell_type_enum get_type() const { return FM_PRISM_CELL; }

  virtual FM_u32 get_n_faces(FM_u32 k) const
  {
    const FM_u32 n_faces[] = {6, 9, 5, 1};
    return k <= 3 ? n_faces[k] : 0;
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    o << "W(" << index;  // "Wedge"
    if (time.defined())
      o <<  ", time=" << time;
    if (submesh_id.defined())
      o <<  ", submesh_id=" << submesh_id;
    return o << ")";
  }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_unstructured_prism* up =
      dynamic_cast<const FM_unstructured_prism*>(&c);
    if (up == 0) return false;
    return
      time == up->time &&
      submesh_id == up->submesh_id &&
      index == up->index;
  }
};


class FM_unstructured_hexahedron : public FM_unstructured_cell
{
public:
  FM_unstructured_hexahedron(const FM_time<FM_u32>& t, const FM_submesh_id& s,
                             FM_u64 i) :
    FM_unstructured_cell(t, s, 3, i) {}

  FM_unstructured_hexahedron(const FM_unstructured_cell& uc, FM_u64 i) :
    FM_unstructured_cell(uc, 3, i) {}

  virtual FM_cell_type_enum get_type() const { return FM_HEXAHEDRON_CELL; }

  virtual FM_u32 get_n_faces(FM_u32 k) const
  {
    const FM_u32 n_faces[] = {8, 12, 6, 1};
    return k <= 3 ? n_faces[k] : 0;
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    o << "H(" << index;
    if (time.defined())
      o <<  ", time=" << time;
    if (submesh_id.defined())
      o <<  ", submesh_id=" << submesh_id;
    return o << ")";
  }

  virtual bool is_equal_to(const FM_cell& c) const
  {
    const FM_unstructured_hexahedron* uh =
      dynamic_cast<const FM_unstructured_hexahedron*>(&c);
    if (uh == 0) return false;
    return
      time == uh->time &&
      submesh_id == uh->submesh_id &&
      index == uh->index;
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
```

```
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_COMBINATORICS_H_
#define _FM_COMBINATORICS_H_
/*
 * NAME: FM_combinatorics.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <vector>
#include "FM_vector.h"

template <typename T>
void FM_swap(T* lhs, T* rhs)
{
  T tmp = *lhs;
  *lhs = *rhs;
  *rhs = tmp;
}

template <typename T>
T FM_fact(T n)
{
  T res = 1;
  for (T i = 2; i <= n; i++)
    res *= i;
  return res;
}

template <typename T>
T FM_choose(T b, T k)
{
  const T LUT_SIZE = 4;
  const char lut[LUT_SIZE][LUT_SIZE] =
  {
    {1, 0, 0, 0},
    {1, 1, 0, 0},
    {1, 2, 1, 0},
    {1, 3, 3, 1}
  };
  // assert(k <= b);
  return b < LUT_SIZE ?
    T(lut[b][k]) : FM_fact(b) / (FM_fact(b - k) * FM_fact(k));
}

template <int B>
void FM_choose_choices(FM_u32 k, std::vector<FM_vector<B,bool> >* choices)
{
  int i, ik = int(k);
  assert(ik <= B);
  FM_u32 n_choices = FM_choose(FM_u32(B), k);
  choices->resize(n_choices);
  int indices[B + 1];
  for (i = 0; i < ik; i++)
    indices[i] = i;
  indices[k] = B;
  FM_vector<B,bool> all_false, choice;
  for (i = 0; i < B; i++)
    all_false[i] = false;

  for (FM_u32 c = 0; c < n_choices; c++) {
    choice = all_false;
    for (i = 0; i < ik; i++)
      choice[indices[i]] = true;
    (*choices)[c] = choice;

    for (i = ik - 1; i >= 0; i--) {
      if (indices[i] + 1 < indices[i + 1]) {
        indices[i]++;
        for (int j = i + 1; j < ik; j++)
          indices[j] = indices[j - 1] + 1;
        break;
      }
    }
  }
}


template <typename T>
inline T FM_pow_2(T i)
{
  return 1 << i;
}

template <typename T>
inline int FM_sign(T i)
{
  return i == T(0) ? 0 : (i > T(0) ? 1 : -1);
}

template <typename T>
inline T FM_abs(T t)
{
  return t >= T(0) ? t : -t;
}

template <typename T>
inline T FM_min(T lhs, T rhs)
{
  return lhs < rhs ? lhs : rhs;
}

template <int N, typename T>
bool FM_odd(const FM_vector<N,T>& v)
{
  T sum = v[0];
  for (int i = 1; i < N; i++)
    sum += v[i];
  return (sum & 1) ? true : false;
}
```

```
/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_CONSTANT_FIELD_H_
#define _FM_CONSTANT_FIELD_H_
/*
 * NAME: FM_constant_field.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_field.h"

template <int B, int D, typename T>
class FM_constant_field : public FM_field<B,D,T>
{
public:
  const T constant;

  FM_constant_field(const FM_ptr<FM_mesh<B,D> >& m, const T& c, int na,
                    FM_properties_cache* pc = 0) :
    FM_field<B,D,T>(m, na, pc),
    constant(c) {}

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_constant_field<" << B << "," << D << ","
             << typeid(T).name() << ">";
  }

  virtual int at_cell(const FM_cell* c, std::vector<T>* vals) const
  {
    std::vector<FM_ptr<FM_cell> > faces =
      mesh->faces(c, node_association_index);
    for (size_t i = 0; i < faces.size(); i++)
      vals->push_back(constant);
    return FM_OK;
  }

  virtual int at_cell(const FM_cell* c, T* vals) const
  {
    std::vector<FM_ptr<FM_cell> > faces =
      mesh->faces(c, node_association_index);
    for (size_t i = 0; i < faces.size(); i++)
      vals[i] = constant;
    return FM_OK;
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_CONTEXT_H_
#define _FM_CONTEXT_H_
/*
 * NAME: FM_context.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_cell.h"


class FM_context
{
public:
  FM_context() :
    simplicial_decomposition(0),
    locate_verbosity(0),
    locate_effort(4)
  {}

  FM_u32 simplicial_decomposition;
  FM_ptr<FM_cell> last_cell;
  FM_u32 locate_verbosity;
  FM_u32 locate_effort;
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_CORE_FIELD_H_
#define _FM_CORE_FIELD_H_
/*
 * NAME: FM_core_field.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_field.h"

template <int B, int D, typename T>
class FM_core_field : public FM_field<B,D,T>
{
public:
  FM_core_field(const FM_ptr<FM_mesh<B,D> >& mesh,
                FM_u32 na, FM_properties_cache* pc) :
    FM_field<B,D,T>(mesh, na, pc) {}

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_core_field<" << B << "," << D << "," <<
      typeid(T).name() << ">";
  }
};


template <int B, int D, typename T>
class FM_core_field_T_layout;

template <int B, int D, typename T>
std::pair<T,T>
FM_get_min_max_aux(const FM_ptr<FM_core_field_T_layout<B,D,T> >& field,
                   const FM_time<FM_u32>* t, const FM_submesh_id* sid,
                   const FM_true_type&);

template <int B, int D, typename T>
class FM_core_field_T_layout : public FM_core_field<B,D,T>
{
public:
  const T* const data;
  const bool delete_suppression;

  FM_core_field_T_layout(const FM_ptr<FM_mesh<B,D> >& m,
                         const T* d, FM_u32 na, bool ds,
                         FM_properties_cache* pc = 0) :
    FM_core_field<B,D,T>(m, na, pc),
    data(d), delete_suppression(ds) {}

  virtual ~FM_core_field_T_layout()
  {
    if (!delete_suppression)
      delete [] data;
  }

  virtual std::pair<T,T> get_min_max(const FM_time<FM_u32>* t = 0,
                                     const FM_submesh_id* sid = 0) const
  {
    return FM_get_min_max_aux(this, t, sid, FM_traits<T>::is_scalar());
  }

  virtual int at_cell(const FM_cell* c, std::vector<T>* vals) const
  {
    std::vector<FM_ptr<FM_cell> > faces =
      mesh->faces(c, node_association_index);
    for (size_t i = 0; i < faces.size(); i++)
      vals->push_back(data[mesh->cell_to_enum(faces[i])]);
    return FM_OK;
  }

  virtual int at_cell(const FM_cell* c, T* vals) const
  {
    std::vector<FM_ptr<FM_cell> > faces =
      mesh->faces(c, node_association_index);
    for (size_t i = 0; i < faces.size(); i++)
      vals[i] = data[mesh->cell_to_enum(faces[i])];
    return  FM_OK;
  }
};


template <int B, int D, typename T>
std::pair<T,T>
FM_get_min_max_aux(const FM_ptr<FM_core_field_T_layout<B,D,T> >& field,
                   const FM_time<FM_u32>* t, const FM_submesh_id* sid,
                   const FM_true_type& tt)
{
  bool blank_checking;
  field->get_simple_value("blank_checking", &blank_checking);
  if (blank_checking)
    return FM_get_min_max_aux(field, t, sid, tt);

  std::pair<T,T> min_max(field->data[0], field->data[0]);
  const T* d = field->data + 1;
  const T* e = field->data +
    field->mesh->get_card(field->node_association_index);
  while (d != e) {
    if (*d < min_max.first)
      min_max.first = *d;
    else if (*d > min_max.second)
      min_max.second = *d;
    d++;
  }
  return min_max;
}


// "Classic" meaning based on structured mesh, and node association index of 0
template <int B, int D, typename T>
class FM_classic_core_field_T_layout : public FM_core_field_T_layout<B,D,T>
```

```
{
public:
  FM_classic_core_field_T_layout
  (const FM_ptr<FM_structured_mesh<B,D> >& m, const T* d, bool ds,
   FM_properties_cache* pc = 0) :
    FM_core_field_T_layout<B,D,T>(m, d, 0, ds, pc)
  {}

  virtual int
  at_cell(const FM_cell* c, std::vector<T>* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[1 << B];
    c->structured_mesh_vertex_indices(mesh, &n_indices, indices);
    if (n_indices == 1)
      vals->push_back(data[indices[0]]);
    else {
      FM_u32 previous_size = vals->size();
      vals->resize(previous_size + n_indices);
      T* dst = &(*vals)[previous_size];
      for (FM_u32 i = 0; i < n_indices; i++) {
        *dst++ = data[indices[i]];
      }
    }
    return FM_OK;
  }

  virtual int
  at_cell(const FM_cell* c, T* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[1 << B];
    c->structured_mesh_vertex_indices(mesh, &n_indices, indices);
    for (FM_u32 i = 0; i < n_indices; i++)
      vals[i] = data[indices[i]];
    return FM_OK;
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_CURVILINEAR_MESH_H_
#define _FM_CURVILINEAR_MESH_H_
/*
 * NAME: FM_curvilinear_mesh.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <algorithm>
#include <queue>
#include "FM_structured_mesh.h"
#include "FM_orient.h"
#include "FM_functional.h"

template <int B, int D> class FM_curvilinear_mesh;


template <int B, int D>
int FM_curvilinear_mesh_adaptive_vertex_walk
(const FM_curvilinear_mesh<B,D>* cm,
 const FM_phys<D>& p,
 const FM_structured_0_cell<B>& initial, FM_structured_0_cell<B>* final)
{
  FM_vector<D,FM_coord> cv;
  FM_structured_0_cell<B> current = initial;
  int res = cm->at_cell(&current, &cv);
  if (res != FM_OK)
    return res;
  FM_coord current_distance = FM_distance2(p, cv);
  FM_coord stride_size = FM_coord(0.1);
  FM_vector<B,FM_u32> stride;
  int d;
  for (d = 0; d < B; d++) {
    stride[d] = FM_u32(FM_coord(cm->dimensions[d]) * stride_size);
    if (stride[d] == 0)
      stride[d] = 1;
  }

  for (;;) {
    bool making_progress = true;

    while (making_progress) {
      FM_coord post_step_distance, best_post_step_distance = current_distance;
      FM_structured_0_cell<B> post_step;
      post_step.set_time(initial.get_time());
      FM_vector<B,FM_u32> best_post_step_indices;

      for (d = 0; d < B; d++) {
        if (stride[d] == 0)
          continue;

        if (stride[d] <= current.get_index(d)) {
          post_step.set_indices(current.get_indices());
          post_step.set_index(d, current.get_index(d) - stride[d]);
          res = cm->at_cell(&post_step, &cv);
          if (res != FM_OK)
            return res;
          post_step_distance = FM_distance2(p, cv);
          if (post_step_distance < best_post_step_distance) {
            best_post_step_distance = post_step_distance;
            best_post_step_indices = post_step.get_indices();
          }
        }

        if (stride[d] + current.get_index(d) < cm->dimensions[d]) {
          post_step.set_indices(current.get_indices());
          post_step.set_index(d, stride[d] + current.get_index(d));
          res = cm->at_cell(&post_step, &cv);
          if (res != FM_OK)
            return res;
          post_step_distance = FM_distance2(p, cv);
          if (post_step_distance < best_post_step_distance) {
            best_post_step_distance = post_step_distance;
            best_post_step_indices = post_step.get_indices();
          }
        }
      }

      making_progress = best_post_step_distance < current_distance;
      if (making_progress) {
        current.set_indices(best_post_step_indices.v());
        current_distance = best_post_step_distance;
      }
    }
    bool has_a_non_zero_stride = false;
    for (d = 0; d < B; d++) {
      stride[d] /= 2;
      if (stride[d] > 0)
        has_a_non_zero_stride = true;
    }
    if (!has_a_non_zero_stride)
      break;
  }
  final->set_time(initial.get_time());
  final->set_submesh_id(initial.get_submesh_id());
  final->set_indices(current.get_indices());
  return FM_OK;
}


//
// The generic FM_curvilinear_mesh<B,D> class.
//
template <int B, int D>
class FM_curvilinear_mesh : public FM_structured_mesh<B,D>
{
protected:
  // Note:  curvilinear_mesh_initialize() needs to be called by
  // the derived classes that define at_cell so this routine
```

```
  // has the option of accessing vertex coordinates as part of the
  // initialization.
  void curvilinear_mesh_initialize()
  {
    FM_vector<B,FM_u32> initial_location = dimensions;
    initial_location /= 2;
    initial_locations.push_back(initial_location);
  }

public:
  FM_curvilinear_mesh(const FM_vector<B,FM_u32>& dimensions,
                      FM_properties_cache* pc = 0) :
    FM_structured_mesh<B,D>(dimensions, pc),
    bounding_box_valid(false)
  {}

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_curvilinear_mesh<" << B << "," << D << ">";
  }

  virtual int
  phys_to_cell(const FM_phys<D>&, FM_context*, FM_ptr<FM_cell>*) const
  {
    FM_ostringstream err;
    err << *this << "::phys_to_cell(): not defined";
    throw std::logic_error(err.str());
  }

  virtual int
  adaptive_vertex_walk(const FM_phys<D>& p,
                       const FM_structured_0_cell<B>& initial,
                       FM_structured_0_cell<B>* final) const
  {
    return FM_curvilinear_mesh_adaptive_vertex_walk(this, p, initial, final);
  }

protected:
  // assume we do not need a mutex for these mutable members
  mutable std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> > bounding_box;
  mutable bool bounding_box_valid;

private:
  std::vector<FM_vector<B,FM_u32> > initial_locations;
};



//
// FM_curvilinear_mesh<3,3> defines hexahedral and tetrahedral walk
// point location methods.
//
template <>
class FM_curvilinear_mesh<3,3> : public FM_structured_mesh<3,3>
{
protected:
  // Note:  curvilinear_mesh_initialize() needs to be called by
  // the derived classes that define at_cell so this routine
  // has the option of accessing vertex coordinates as part of the
  // initialization.
  void curvilinear_mesh_initialize()
  {
    // choose initial positions for starting searches for a close vertex
    // choose center of each of the 6 mesh faces as FEL1 does
    FM_u32 dim0_050 = (dimensions[0] - 1) / 2;
    FM_u32 dim0m1 = dimensions[0] - 1;
    FM_u32 dim1_050 = (dimensions[1] - 1) / 2;
    FM_u32 dim1m1 = dimensions[1] - 1;
    FM_u32 dim2_050 = (dimensions[2] - 1) / 2;
    FM_u32 dim2m1 = dimensions[2] - 1;

    int d;
    FM_vector<3,FM_u32> initial_location;
    for (d = 0; d < 3; d++)
      initial_location[d] = dimensions[d] / 2;
    initial_locations.push_back(initial_location);
    initial_locations.push_back(FM_vector<3,FM_u32>(dim0_050, dim1_050, 1));
    initial_locations.push_back
      (FM_vector<3,FM_u32>(dim0_050, dim1_050, dim2m1));
    initial_locations.push_back(FM_vector<3,FM_u32>(dim0_050, 1, dim2_050));
    initial_locations.push_back
      (FM_vector<3,FM_u32>(dim0_050, dim1m1, dim2_050));
    initial_locations.push_back(FM_vector<3,FM_u32>(1, dim1_050, dim2_050));
    initial_locations.push_back
      (FM_vector<3,FM_u32>(dim0m1, dim1_050, dim2_050));
  }

public:
  FM_curvilinear_mesh(const FM_vector<3,FM_u32>& dimensions,
                      FM_properties_cache* pc = 0) :
    FM_structured_mesh<3,3>(dimensions, pc),
    bounding_box_valid(false)
  {}

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_curvilinear_mesh<3,3>";
  }

  virtual int
  adaptive_vertex_walk(const FM_phys<3>& p,
                       const FM_structured_0_cell<3>& initial,
                       FM_structured_0_cell<3>* final) const
  {
    return FM_curvilinear_mesh_adaptive_vertex_walk(this, p, initial, final);
  }

  int hexahedral_walk_locate(const FM_phys<3>& p,
                             const FM_ptr<FM_structured_cell<3> >& initial,
                             FM_context* ctxt,
                             FM_ptr<FM_cell>* c) const
{
  FM_ostringstream verbose_prefix;
  if (ctxt->locate_verbosity > 0) {
    verbose_prefix << *this << "::hexahedral_walk_locate(" << p << ", " <<
      initial << ",,): ";
    std::cout << verbose_prefix.str() << std::endl;
  }

  assert(initial->get_dimension() == 3);
  FM_vector<3,FM_u32> indices = initial->get_indices();

  FM_u32 face = 0;
  FM_u32 faces_tested = 0;
  FM_u32 total_faces_tested = 0;
  FM_u32 total_faces_tested_threshold =
    2 * (dimensions[0] + dimensions[1] + dimensions[2]);
  bool suppressed_step_off_mesh = false;
  FM_vector<3,FM_coord> cv[8];
  FM_u32 even_odd =
    FM_simplicial_decomposition_odd(indices,
                                    ctxt->simplicial_decomposition) ? 1 : 0;
  int res;
  while (faces_tested < 6) {
    total_faces_tested += 1;
    if (total_faces_tested > total_faces_tested_threshold) {
      if (ctxt->locate_verbosity > 0)
        std::cout << verbose_prefix.str() <<
          "stuck, giving up" << std::endl;
      return FM_POINT_LOCATE_STUCK;
    }

    *c = new FM_structured_B_cell<3>(initial->get_time(),
                                     initial->get_submesh_id(),
                                     indices);
    res = at_cell(*c, cv);
    if (res != FM_OK) return res;

    int orientation =
      FM_orient(cv[FM_hexahedron_face[even_odd][face][0]],
                cv[FM_hexahedron_face[even_odd][face][1]],
                cv[FM_hexahedron_face[even_odd][face][2]],
                cv[FM_hexahedron_face[even_odd][face][3]],
                p);

    bool outside = orientation < 0;

    if (ctxt->locate_verbosity > 1)
      std::cout << verbose_prefix.str() << **c << ", face: " << face <<
        ", " << FM_orientation_names[orientation + 1] << std::endl;

    if (outside) {
      // step to next hexahedron, if step does not take us off mesh
      FM_u32 axis = face / 2;
      bool low_side = !bool(face & 1);
      if (low_side) {
        if (indices[axis] == 0) {
          suppressed_step_off_mesh = true;
          goto next_hexahedron_face;
        }
        indices[axis] -= 1;
      }
      else {
        if (indices[axis] == dimensions[axis] - 2) {
          suppressed_step_off_mesh = true;
          goto next_hexahedron_face;
        }
        indices[axis] += 1;
      }
      even_odd ^= 1;
      face = face ^ 1;
      faces_tested = 0;
      suppressed_step_off_mesh = false;
    }
  next_hexahedron_face:
    face = (face + 1) % 6;
    faces_tested += 1;
  }

  ctxt->last_cell = *c;

  if (!suppressed_step_off_mesh) {
    res = FM_OK;
    if (ctxt->simplicial_decomposition) {
      FM_ptr<FM_structured_B_cell<3> > sc = *c;
      res = phys_to_subsimplex(p, sc, ctxt, c);
    }
  }
  else
    res = FM_POINT_LOCATE_WALKED_OFF_MESH;

  if (ctxt->locate_verbosity > 1)
    std::cout << verbose_prefix.str() << "tested " << total_faces_tested <<
      " quadrilaterals" << std::endl;
  if (ctxt->locate_verbosity > 0)
    std::cout << verbose_prefix.str() << **c << ", returning " <<
      res << std::endl;

  return res;
}


int tetrahedral_walk_locate(const FM_phys<3>& p,
                            const FM_ptr<FM_structured_cell<3> >& initial,
                            FM_context* ctxt,
                            FM_ptr<FM_cell>* c) const
{
  FM_ostringstream verbose_prefix;
  if (ctxt->locate_verbosity > 0) {
    verbose_prefix << *this << "::tetrahedral_walk_locate(" << p << ", " <<
```

```
    *initial << ",,): ";
  std::cout << verbose_prefix.str() << std::endl;
}

assert(initial->get_dimension() == 3);
FM_vector<3,FM_u32> indices = initial->get_indices();
FM_u32 subid;
if (initial->is_subsimplex())
  subid = initial->get_subid();
else
  subid =
    !FM_simplicial_decomposition_odd
    (indices, ctxt->simplicial_decomposition) ? 4 : 9;
bool new_cell = true;

FM_u32 face = 0;
FM_u32 faces_tested = 0;
FM_u32 total_faces_tested = 0;
FM_u32 total_faces_tested_threshold =
  4 * 5 * (dimensions[0] + dimensions[1] + dimensions[2]);
bool suppressed_step_off_mesh = false;
FM_vector<3,FM_coord> cv[8];
int res;
while (faces_tested < 4) {
  total_faces_tested += 1;
  if (total_faces_tested > total_faces_tested_threshold) {
    if (ctxt->locate_verbosity > 0)
      std::cout << verbose_prefix.str() <<
        "stuck, trying hexahedral_walk_locate" << std::endl;
    return hexahedral_walk_locate(p, initial, ctxt, c);
  }

  if (new_cell) {
    *c = new FM_structured_B_cell<3>(initial->get_time(),
                                     initial->get_submesh_id(),
                                     indices);
    res = at_cell(*c, cv);
    if (res != FM_OK) return res;
    new_cell = false;
  }

  int orientation =
    FM_orient(cv[FM_subtetrahedron_face[subid][face][0]],
              cv[FM_subtetrahedron_face[subid][face][1]],
              cv[FM_subtetrahedron_face[subid][face][2]],
              p);

  bool outside = orientation < 0;

  if (ctxt->locate_verbosity > 1)
    std::cout << verbose_prefix.str() << **c << ", subid: " <<
      subid << ", face: " << face << ", " <<
      FM_orientation_names[orientation + 1] << std::endl;

  if (orientation == 0)
    return hexahedral_walk_locate(p, *c, ctxt, c);

  if (outside) {
    // step to next tetrahedron, if step does not take us off mesh
    FM_u32 axis = FM_tetrahedron_step[subid][face].axis;
    switch(FM_tetrahedron_step[subid][face].dir) {
    case 0:
      break;
    case -1:
      if (indices[axis] == 0) {
        suppressed_step_off_mesh = true;
        goto next_subtetrahedron_face;
      }
      indices[axis] -= 1;
      new_cell = true;
      break;
    case 1:
      if (indices[axis] == dimensions[axis] - 2) {
        suppressed_step_off_mesh = true;
        goto next_subtetrahedron_face;
      }
      indices[axis] += 1;
      new_cell = true;
      break;
    default:
      abort();
    }
    FM_u32 prev_subid = subid;
    FM_u32 prev_face = face;
    subid = FM_tetrahedron_step[prev_subid][prev_face].subsimplex;
    face = FM_tetrahedron_step[prev_subid][prev_face].subsimplex_face;
    faces_tested = 0;
    suppressed_step_off_mesh = false;
  }
next_subtetrahedron_face:
  face = (face + 1) & 3;
  faces_tested += 1;
}

if (!suppressed_step_off_mesh) {
  if (ctxt->simplicial_decomposition)
    *c = new FM_structured_subsimplex<3>((*c)->get_time(),
                                         (*c)->get_submesh_id(),
                                         3, subid, indices);
  res = FM_OK;
}
else
  res = FM_POINT_LOCATE_WALKED_OFF_MESH;

ctxt->last_cell = *c;

if (ctxt->locate_verbosity > 1)
  std::cout << verbose_prefix.str() << "tested " << total_faces_tested <<
    " triangles" << std::endl;
if (ctxt->locate_verbosity > 0)
    std::cout << verbose_prefix.str() << **c << ", returning " <<
      res << std::endl;

  return res;
}

virtual int
phys_to_cell(const FM_phys<3>& p, FM_context* ctxt, FM_ptr<FM_cell>* c) const
{
  FM_ostringstream verbose_prefix;
  if (ctxt->locate_verbosity > 0) {
    verbose_prefix << *this << "::phys_to_cell(" << p << ",,): ";
    std::cout << verbose_prefix.str() << std::endl;
  }

  int res;
  FM_u32 axis;

  while (1) {
    // if we have a previous cell, try walking from there
    if (ctxt->last_cell) {
      if (ctxt->locate_verbosity > 0) {
        std::cout << verbose_prefix.str() << "starting from last_cell " <<
          *ctxt->last_cell << std::endl;
      }
      res = tetrahedral_walk_locate(p, ctxt->last_cell, ctxt, c);
      if (res == FM_OK) break;
    }

    // first test against bounding box
    if (!bounding_box_valid)
      (void) get_bounding_box();
    bool outside = false;
    for (axis = 0; axis < 3; axis++) {
      if (!(bounding_box.first[axis] <= p[axis] &&
            p[axis] <= bounding_box.second[axis])) {
        res = FM_POINT_OUTSIDE_BOUNDING_BOX;
        outside = true;
        break;
      }
    }
    if (outside) break;

    // put initial locations in priority queue (pq), ordered by
    // distance to p (closest to farthest); adaptive vertex walk,
    // then tetrahedral walk from each unique adaptive walk destination
    // until success or queue is empty
    typedef std::pair<FM_coord,FM_vector<3,FM_u32> > pq_element;
    typedef std::vector<pq_element> pq_impl;
    typedef FM_first_greater_pred<pq_element> pq_pred;
    std::priority_queue<pq_element,pq_impl,pq_pred>
      initial_locations_priority_queue;
    FM_structured_0_cell<3> v;
    // v.set_time
    // fill priority queue
    for (FM_u32 i = 0; i < initial_locations.size(); i++) {
      v.set_indices(initial_locations[i].v());
      FM_vector<3,FM_coord> cv;
      res = at_cell(&v, &cv);
      if (res != FM_OK) break;
      FM_coord d2 = FM_distance2(p, cv);
      initial_locations_priority_queue.push
        (std::make_pair(d2, initial_locations[i]));
    }
    if (res != FM_OK) break;

    // walk from each location in queue
    std::vector<FM_vector<3,FM_u32> > adaptive_walk_destinations;
    while (!initial_locations_priority_queue.empty()) {
      v.set_indices(initial_locations_priority_queue.top().second.v());
      initial_locations_priority_queue.pop();
      FM_structured_0_cell<3> adaptive_walk_destination;
      res = adaptive_vertex_walk(p, v, &adaptive_walk_destination);
      if (res != FM_OK) break;

      // do not repeat tetrahedral walk if already done from this
      // adaptive vertex walk destination -- i.e, been there, done that
      if (std::find(adaptive_walk_destinations.begin(),
                    adaptive_walk_destinations.end(),
                    adaptive_walk_destination.get_indices()) !=
          adaptive_walk_destinations.end())
        continue;
      adaptive_walk_destinations.push_back
        (adaptive_walk_destination.get_indices());

      // step back if necessary from boundary before constructing hexahedron
      FM_vector<3,FM_u32> indices = adaptive_walk_destination.get_indices();
      for (axis = 0; axis < 3; axis++) {
        if (indices[axis] == dimensions[axis] - 1)
          indices[axis]--;
      }
      FM_ptr<FM_structured_cell<3> > initial_cell =
        new FM_structured_B_cell<3>(indices);

      res = tetrahedral_walk_locate(p, initial_cell, ctxt, c);
      if (res == FM_OK) break;
    }

    break;
  }

  if (ctxt->locate_verbosity > 0)
    std::cout << verbose_prefix.str() << "returning " << res << std::endl;
  return res;
}

protected:
  // assume we do not need a mutex for these mutable members
  mutable std::pair<FM_vector<3,FM_coord>,FM_vector<3,FM_coord> > bounding_box;
  mutable bool bounding_box_valid;
```

```
private:
  std::vector<FM_vector<3,FM_u32> > initial_locations;
};


//
// FM_curvilinear_mesh_T_layout<B,D> is a curvilinear mesh where the
// coordinates are contained in a single array of FM_vector<D,FM_coord>,
// i.e., an array where the coordinates for each vertex are contiguous.
//
template <int B, int D>
class FM_curvilinear_mesh_T_layout : public FM_curvilinear_mesh<B,D>
{
public:
  const FM_vector<D,FM_coord>* const coordinates;
  const bool delete_suppression;

  FM_curvilinear_mesh_T_layout(const FM_vector<B,FM_u32>& dimensions,
                               const FM_vector<D,FM_coord>* c, bool ds,
                               FM_properties_cache* pc = 0) :
    FM_curvilinear_mesh<B,D>(dimensions, pc),
    coordinates(c),
    delete_suppression(ds)
  {
    curvilinear_mesh_initialize();
  }

  ~FM_curvilinear_mesh_T_layout()
  {
    if (!delete_suppression)
      delete [] coordinates;
  }

  virtual std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> >
  get_bounding_box(const FM_time<FM_u32>* = 0,
                   const FM_submesh_id* = 0) const;

  virtual int
  at_cell(const FM_cell* c, std::vector<FM_vector<D,FM_coord> >* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[1 << B];
    c->structured_mesh_vertex_indices(this, &n_indices, indices);
    if (n_indices == 1)
      vals->push_back(coordinates[indices[0]]);
    else {
      FM_u32 previous_size = vals->size();
      vals->resize(previous_size + n_indices);
      FM_vector<D,FM_coord>* dst = &(*vals)[previous_size];
      for (FM_u32 i = 0; i < n_indices; i++) {
        *dst++ = coordinates[indices[i]];
      }
    }
    return FM_OK;
  }

  virtual int
  at_cell(const FM_cell* c, FM_vector<D,FM_coord>* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[1 << B];
    c->structured_mesh_vertex_indices(this, &n_indices, indices);
    for (FM_u32 i = 0; i < n_indices; i++)
      vals[i] = coordinates[indices[i]];
    return FM_OK;
  }
};


//
// FM_curvilinear_mesh_T_layout<B,D>::get_bounding_box works with
// a pointer directly into the coordinates buffer, testing every vertex.
//
template <int B, int D>
std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> >
FM_curvilinear_mesh_T_layout<B,D>::
get_bounding_box(const FM_time<FM_u32>*, const FM_submesh_id*) const
{
  if (!bounding_box_valid) {
    std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> > bb;
    const FM_vector<D,FM_coord>* cp = coordinates;
    bb.first = *cp;
    bb.second = *cp;
    cp++;
    FM_u64 card0 = get_card(0);
    for (FM_u64 i = 1; i < card0; i++)
      FM_operator_min_max_equals(bb, *cp++);
    bounding_box = bb;
    bounding_box_valid = true;
  }
  return bounding_box;
}


//
// FM_curvilinear_mesh_T_layout<3,3>::get_bounding_box works with
// a pointer directly into the coordinates buffer, testing vertices
// on the boundary of the mesh.
//
template <>
std::pair<FM_vector<3,FM_coord>,FM_vector<3,FM_coord> >
FM_curvilinear_mesh_T_layout<3,3>::
get_bounding_box(const FM_time<FM_u32>*, const FM_submesh_id*) const
{
  if (!bounding_box_valid) {
    FM_u32 i, j, k;
    std::pair<FM_vector<3,FM_coord>,FM_vector<3,FM_coord> > bb;
    const FM_vector<3,FM_coord>* cp = coordinates;
    bb.first = *cp;
    bb.second = *cp;
    cp++;

    // k = 0 slice
    FM_u32 dimensions_0_dimensions_1 = dimensions[0] * dimensions[1];
    for (i = 1; i < dimensions_0_dimensions_1; i++)
      FM_operator_min_max_equals(bb, *cp++);

    // edge of k = 1 to dimensions[2] - 2 slices
    for (k = 1; k < dimensions[2] - 1; k++) {
      for (i = 0; i < dimensions[0]; i++)
        FM_operator_min_max_equals(bb, *cp++);
      for (j = 1; j < dimensions[1] - 1; j++) {
        FM_operator_min_max_equals(bb, *cp);
        cp += dimensions[0] - 1;
        FM_operator_min_max_equals(bb, *cp++);
      }
      for (i = 0; i < dimensions[0]; i++)
        FM_operator_min_max_equals(bb, *cp++);
    }
    // k = dimensions[2] - 1 slice
    for (i = 0; i < dimensions_0_dimensions_1; i++)
      FM_operator_min_max_equals(bb, *cp++);

    bounding_box = bb;
    bounding_box_valid = true;
  }
  return bounding_box;
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_DERIVED_FIELD_H_
#define _FM_DERIVED_FIELD_H_
/*
 * NAME: FM_derived_field.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_field.h"


template <int B, int D, typename S, typename T, typename F>
class FM_unary_derived_field : public FM_field<B,D,T>
{
public:
  const FM_ptr<FM_shared_object> so;
  const FM_field_interface<B,D,S>* field;
  const F function;

  FM_unary_derived_field(const FM_ptr<FM_field<B,D,S> >& f,
                         const F& fun, FM_properties_cache* pc = 0) :
    FM_field<B,D,T>(f->mesh, f->node_association_index, pc),
    so(f),
    field(f),
    function(fun)
  {}

  FM_unary_derived_field(const FM_ptr<FM_mesh<B,D> >& m,
                         const F& fun, FM_properties_cache* pc = 0) :
    FM_field<B,D,T>(m, 0, pc),
    so(m),
    field(m),
    function(fun)
  {}

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_unary_derived_field<" <<
      B << "," << D << "," << typeid(S).name() << "," <<
      typeid(T).name() << "," << typeid(F).name() << ">";
  }

  virtual int at_cell(const FM_cell* c, std::vector<T>* vals) const
  {
    std::vector<S> tmp;
    int res = field->at_cell(c, &tmp);
    if (res != FM_OK) return res;
    FM_u32 previous_size = vals->size();
    vals->resize(previous_size + tmp.size());
    T* dst = &(*vals)[previous_size];
    for (size_t i = 0; i < tmp.size(); i++) {
      res = function(tmp[i], dst++);
      if (res != FM_OK) break;
    }
    return res;
  }

  virtual int at_cell(const FM_cell* c, T* vals) const
  {
    std::vector<S> tmp;
    int res = field->at_cell(c, &tmp);
    if (res != FM_OK) return res;
    for (size_t i = 0; i < tmp.size(); i++) {
      res = function(tmp[i], &vals[i]);
      if (res != FM_OK) break;
    }
    return res;
  }
};


template <int B, int D, typename R, typename S, typename T, typename F>
class FM_binary_derived_field : public FM_field<B,D,T>
{
public:
  const FM_ptr<FM_shared_object> lhs_so;
  const FM_ptr<FM_shared_object> rhs_so;
  const FM_field_interface<B,D,R>* lhs_field;
  const FM_field_interface<B,D,S>* rhs_field;
  const F function;

  FM_binary_derived_field(const FM_ptr<FM_field<B,D,R> >& lhs,
                          const FM_ptr<FM_field<B,D,S> >& rhs,
                          const F& fun,
                          FM_properties_cache* pc = 0) :
    FM_field<B,D,T>(lhs->mesh, lhs->node_association_index, pc),
    lhs_so(lhs),
    rhs_so(rhs),
    lhs_field(lhs),
    rhs_field(rhs),
    function(fun)
  {
    init();
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_binary_derived_field<" << B << "," << D << "," <<
      typeid(R).name() << "," << typeid(S).name() << "," <<
      typeid(T).name() << "," << typeid(F).name() << ">";
  }

  virtual int at_cell(const FM_cell* c, std::vector<T>* vals) const
  {
    std::vector<R> lhs_tmp;
    int res;
    res = lhs_field->at_cell(c, &lhs_tmp);
    if (res != FM_OK) return res;
    std::vector<S> rhs_tmp(lhs_tmp.size());
    rhs_tmp.clear();
    res = rhs_field->at_cell(c, &rhs_tmp);
    if (res != FM_OK) return res;
    FM_u32 previous_size = vals->size();
    vals->resize(previous_size + lhs_tmp.size());
    for (size_t i = 0; i < lhs_tmp.size(); i++) {
      res = function(lhs_tmp[i], rhs_tmp[i], &(*vals)[previous_size + i]);
      if (res != FM_OK) break;
    }
    return res;
  }

  virtual int at_cell(const FM_cell* c, T* vals) const
  {
    return -1;
  }


private:
  void init() const {
    if (lhs_field->mesh != rhs_field->mesh) {
      FM_ostringstream err;
      err << "FM_binary_derived_field(";
      err << *lhs_so << ", " << *rhs_so << ", ): ";
      err << "fields must be based on same mesh";
      throw std::logic_error(err.str());
    }

    if (lhs_field->node_association_index !=
        rhs_field->node_association_index) {
      FM_ostringstream err;
      err << "FM_binary_derived_field(";
      err << *lhs_so << ", " << *rhs_so << ", ): ";
      err << "node association indices must match (node_association_indices: ";
      err << lhs_field->node_association_index << ", ";
      err << rhs_field->node_association_index << ")";
      throw std::logic_error(err.str());
    }
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_FIELD_H_
#define _FM_FIELD_H_
/*
 * NAME: FM_field.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <typeinfo>
#include <string>
#include <vector>
#include "FM_shared_object_with_properties_cache.h"
#include "FM_field_interface.h"
#include "FM_mesh.h"
#include "FM_ostringstream.h"

#include "FM_structured_mesh.h"

std::string FM_field_type_name(FM_u32 b, FM_u32 d, const char* t)
{
  FM_ostringstream os;
  os << "field<" << b << "," << d << "," << t << ">";
  return os.str();
}


class FM_field_ : public FM_shared_object_with_properties_cache
{
public:
  const FM_ptr<FM_mesh_> mesh_;
  const char* const node_type;
  const bool structured_behavior;

  FM_field_(const FM_ptr<FM_mesh_>& m, const char* nt,
            FM_properties_cache* pc) :
    FM_shared_object_with_properties_cache(pc),
    mesh_(m),
    node_type(nt),
    structured_behavior(mesh_->get_structured_behavior())
  {}

  inline FM_iter begin() const { return mesh_->begin(); }
  inline FM_iter begin(const FM_iter_attrs& ia) const
  {
    return mesh_->begin(ia);
  }
  inline FM_iter end() const { return mesh_->end(); }

protected:
  virtual FM_ptr<FM_shared_object>
  get_aux(const std::string& key, FM_u32 pass,
          const FM_time<FM_u32>* t, const FM_submesh_id* sid) const
  {
    if (key == "node_type") {
      return new FM_simple_value<std::string>(node_type);
    }
    else if (key == "field_type_name") {
      return new FM_simple_value<std::string>
        (FM_field_type_name(mesh_->base_dimensionality,
                            mesh_->phys_dimensionality,
                            node_type));
    }
    else if (key == "mesh") {
      return mesh_;
    }

    FM_ptr<FM_shared_object> mesh_property;
    try {
      mesh_property = mesh_->get(key, t, sid);
    }
    catch (...) {
    }
    if (mesh_property != 0)
      return mesh_property;

    return FM_shared_object_with_properties_cache::
      get_aux(key, pass, t, sid);
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& property_names,
                         const FM_time<FM_u32>* t,
                         const FM_submesh_id* sid) const
  {
    property_names.insert("node_type");
    property_names.insert("field_type_name");
    property_names.insert("mesh");

    std::set<std::string> mesh_property_names =
      mesh_->get_property_names(t, sid);
    std::set<std::string>::const_iterator iter;
    for (iter = mesh_property_names.begin();
         iter != mesh_property_names.end(); ++iter)
      property_names.insert(*iter);

    return FM_shared_object_with_properties_cache::
      get_property_names_aux(property_names, t, sid);
  }
};


template <int B, int D, typename T>
class FM_field;

template <int B, int D, typename T, typename S>
std::pair<T,T>
FM_get_min_max_aux(const FM_field<B,D,T>*, const FM_time<FM_u32>*,
                   const FM_submesh_id*, const S&);


template <int B, int D, typename T>
class FM_field : public FM_field_, public FM_field_interface<B,D,T>
{
public:
  FM_field(const FM_ptr<FM_mesh<B,D> >& m, FM_u32 na,
           FM_properties_cache* pc) :
    FM_field_(m, typeid(T).name(), pc),
    FM_field_interface<B,D,T>(m, na)
  {
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << FM_field_type_name(B, D, typeid(T).name());
  }

  virtual std::pair<T,T> get_min_max(const FM_time<FM_u32>* t = 0,
                                     const FM_submesh_id* sid = 0) const
  {
    return FM_get_min_max_aux(this, t, sid, FM_traits<T>::is_scalar());
  }

  virtual int at_phys(const FM_phys<D>& p, FM_context* ctxt, T* val) const
  {
    int res;
    if (structured_behavior && node_association_index == 0 &&
        ctxt->simplicial_decomposition == 0) {
      FM_base<B> b;
      FM_ptr<FM_structured_B_cell<B> > sc;
      res = mesh->phys_to_base(p, ctxt, &b, &sc);
      if (res != FM_OK) return res;
      T vals[1 << B];
      res = at_cell(sc, vals);
      if (res != FM_OK) return res;
      res = FM_linear_interpolate(b, sc, vals, val);
    }
    else {
      FM_ptr<FM_cell> c;
      res = mesh->phys_to_cell(p, ctxt, &c);
      if (res != FM_OK) return res;
      std::vector<T> vals;
      res = at_cell(c, &vals);
      if (res != FM_OK) return res;

      // interpolate ...
      res = -1;
    }
    return res;
  }

protected:
  virtual FM_ptr<FM_shared_object>
  get_aux(const std::string& key, FM_u32 pass,
          const FM_time<FM_u32>* t, const FM_submesh_id* sid) const
  {
    if (key == "node_association_index") {
      return new FM_simple_value<FM_u32>(node_association_index);
    }
    else if (key == "min_max") {
      std::pair<T,T> min_max = get_min_max(t, sid);
      return new FM_tuple_value(new FM_simple_value<T>(min_max.first),
                                new FM_simple_value<T>(min_max.second));
    }
    return FM_field_::get_aux(key, pass, t, sid);
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& property_names,
                         const FM_time<FM_u32>* t,
                         const FM_submesh_id* sid) const
  {
    property_names.insert("node_association_index");
    property_names.insert("min_max");
    return FM_field_::get_property_names_aux(property_names, t, sid);
  }
};


template <int B, int D, typename T, typename S>
std::pair<T,T>
FM_get_min_max_aux(const FM_field<B,D,T>* field,
                   const FM_time<FM_u32>* t,
                   const FM_submesh_id* sid, const S&)
{
  FM_ostringstream err;
  err << *field << "::get_min_max(" <<
    (t ? *t : FM_time<FM_u32>()) << ", " <<
    (sid ? *sid : FM_submesh_id()) << "): ";
  err << "not defined for node type " << typeid(T).name();
  throw std::logic_error(err.str());
}


template <int B, int D, typename T>
std::pair<T,T>
FM_get_min_max_aux(const FM_field<B,D,T>* field,
                   const FM_time<FM_u32>* t, const FM_submesh_id* sid,
                   const FM_true_type&)
{
  std::pair<T,T> min_max;
  FM_iter_attrs iter_attrs;
  iter_attrs.push_back(new FM_cell_dimension_iter_attr
                       (field->node_association_index));
  if (t && t->defined())
    iter_attrs.push_back(new FM_time_iter_attr(*t));
  if (sid && sid->defined())
    iter_attrs.push_back(new FM_submesh_id_iter_attr(*sid));
  FM_iter i = field->begin(iter_attrs);
  FM_iter e = field->end();
```

```
  std::vector<T> vals(1);
  for ( ; i != e; ++i) {
    vals.clear();
    int res = field->at_cell(*i, &vals);
    if (res == FM_OK) break;
  }
  if (i == e) {
    FM_ostringstream err;
    err << *field << "::get_min_max(" <<
      (t ? *t : FM_time<FM_u32>()) << ", " <<
      (sid ? *sid : FM_submesh_id()) <<
      "): no valid values";
    throw std::logic_error(err.str());
  }
  min_max.first = vals[0];
  min_max.second = vals[0];
  ++i;
  for ( ; i != e; ++i) {
    vals.clear();
    int res = field->at_cell(*i, &vals);
    if (res != FM_OK) continue;
    if (vals[0] < min_max.first)
      min_max.first = vals[0];
    else if (vals[0] > min_max.second)
      min_max.second = vals[0];
  }
  return min_max;
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_FIELD_INTERFACE_H_
#define _FM_FIELD_INTERFACE_H_
/*
 * NAME: FM_field_interface.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <vector>
#include "FM_phys.h"
#include "FM_base.h"
#include "FM_context.h"
#include "FM_cell.h"
#include "FM_shared_object.h"


template <int B, int D>
class FM_mesh;

template <int B, int D, typename T>
class FM_field_interface
{
public:
  const FM_mesh<B,D>* const mesh;
  const FM_u32 node_association_index;

  FM_field_interface(const FM_mesh<B,D>* m, FM_u32 nai) :
    mesh(m), node_association_index(nai) {}

  virtual std::ostream& str(std::ostream& o) const = 0;

  virtual int at_base(const FM_base<B>& b, FM_context*, T* val) const
  {
    int res;
    FM_ptr<FM_structured_B_cell<B> > sc;
    res = mesh->base_to_cell(b, &sc);
    if (res != FM_OK) return res;
    T vals[1 << B];
    res = at_cell(sc, vals);
    if (res != FM_OK) return res;
    return FM_linear_interpolate(b, sc, vals, val);
  }

  virtual int at_base(const FM_base<B,FM_u32>& b, T* val) const
  {
    FM_structured_0_cell<B> sc(b);
    return at_cell(&sc, val);
  }

  virtual int at_phys(const FM_phys<D>&, FM_context*, T*) const = 0;

  virtual int at_cell(const FM_cell*, std::vector<T>*) const = 0;
  virtual int at_cell(const FM_cell*, T*) const = 0;

  virtual int blanks_at_cell(const FM_cell* c, std::vector<int>* blanks) const
  {
    std::vector<T> vals;
    int res = at_cell(c, &vals);
    for (size_t i = 0; i < vals.size(); i++)
      blanks->push_back(1);
    return res;
  }
};


void FM_throw_bad_cell_argument(const FM_shared_object* so,
                                const char* method, const FM_cell* c)
{
  FM_ostringstream err;
  err << *so << "::" << method << "(" << *c << ",...): bad cell argument type";
  throw std::logic_error(err.str());
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_FUNCTIONAL_H_
#define _FM_FUNCTIONAL_H_
/*
 * NAME: FM_functional.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <functional>
#include "FM_ostringstream.h"
#include "FM_vector.h"
#include "FM_shared_object.h"

//
// from <functional>:
//   plus, minus, multiplies, divides, modulus, negate,
//   equal_to, not_equal_to, greater, less, greater_equal, less_equal,
//   logical_and, logical_or, logical_not
//


template <typename T>
class FM_negate_fun : public std::unary_function<T,T>
{
public:
  int operator()(const T& t, T* res) const
  {
    *res =  -t;
    return FM_OK;
  }
};


template <typename T>
class FM_plus_fun : public std::binary_function<T,T,T>
{
public:
  int operator()(const T& lhs, const T& rhs, T* res) const
  {
    *res = lhs + rhs;
    return FM_OK;
  }
};


template <typename T>
class FM_minus_fun : public std::binary_function<T,T,T>
{
public:
  int operator()(const T& lhs, const T& rhs, T* res) const
  {
    *res = lhs - rhs;
    return FM_OK;
  }
};


template <typename T>
class FM_multiplies_fun : public std::binary_function<T,T,T>
{
public:
  int operator()(const T& lhs, const T& rhs, T* res) const
  {
    *res = lhs * rhs;
    return FM_OK;
  }
};


template <typename T>
class FM_divides_fun : public std::binary_function<T,T,T>
{
public:
  int operator()(const T& lhs, const T& rhs, T* res) const
  {
    *res = lhs / rhs;
    return FM_OK;
  }
};


template <typename S, typename T>
class FM_fun_ptr_fun : public std::unary_function<S,T>
{
public:
  FM_fun_ptr_fun(const FM_shared_object* ctxt,
                 int (*f)(const FM_shared_object*, const S&, T*)) :
    fun_ctxt(ctxt), fun(f) {}

  int operator()(const S& s, T* t) const { return fun(fun_ctxt, s, t); }

private:
  FM_ptr<FM_shared_object> fun_ctxt;
  int (*fun)(const FM_shared_object*, const S&, T*);
};


template <typename T>
class FM_abs_fun : public std::unary_function<T,T>
{
public:
  int operator()(const T& t, T* res) const
  {
    *res =  t >= T(0) ? t : -t;
    return FM_OK;
  }
};


template <typename S, typename T>
class FM_static_cast_fun : public std::unary_function<S,T>
{
public:
  int operator()(const S& s, T* t) const
  {
    *t = static_cast<T>(s);
    return FM_OK;
  }
};


template <typename T>
class FM_min_fun : public std::binary_function<T,T,T>
{
public:
  int operator()(const T& lhs, const T& rhs, T* res) const
  {
    *res = lhs < rhs ? lhs : rhs;
    return FM_OK;
  }
};


template <typename T>
class FM_max_fun : public std::binary_function<T,T,T>
{
public:
  int operator()(const T& lhs, const T& rhs, T* res) const
  {
    *res = lhs > rhs ? lhs : rhs;
    return FM_OK;
  }
};


template <int N, typename T>
class FM_dot_fun : public std::binary_function<FM_vector<N,T>,FM_vector<N,T>,T>
{
public:
  int operator()(const FM_vector<N,T>& lhs, const FM_vector<N,T>& rhs,
                 T* res) const
  {
    *res = FM_dot(lhs, rhs);
    return FM_OK;
  }
};


template <typename T>
class FM_cross_fun :
  public std::binary_function<FM_vector<3,T>,FM_vector<3,T>,FM_vector<3,T> >
{
public:
  int operator()(const FM_vector<3,T>& lhs, const FM_vector<3,T>& rhs,
                 FM_vector<3,T>* res) const
  {
    *res = FM_cross(lhs, rhs);
    return FM_OK;
  }
};


template <int N, typename T>
class FM_mag_fun : public std::unary_function<FM_vector<N,T>,T>
{
public:
  int operator()(const FM_vector<N,T>& v, T* res) const
  {
    *res = FM_mag(v);
    return FM_OK;
  }
};


template <int N, typename T>
class FM_brackets_fun : public std::unary_function<FM_vector<N,T>,T>
{
public:
  FM_brackets_fun(int i) : index(i)
  {
    if (!(0 <= index && index < N)) {
      FM_ostringstream err;
      err << "FM_brackets_fun<" << N << ",T>::FM_brackets_fun(" <<
        i << "): bad index";
      throw std::logic_error(err.str());
    }
  }

  int operator()(const FM_vector<N,T>& v, T* res) const
  {
    *res = v[index];
    return FM_OK;
  }

private:
  const int index;
};


template <int M, int N, typename T>
class FM_slice_brackets_fun :
  public std::unary_function<FM_vector<M,T>,FM_vector<N,T> >
{
public:
  FM_slice_brackets_fun(int i) : index(i)
  {
    if (!(0 <= index && index + N <= M)) {
      FM_ostringstream err;
      err << "FM_slice_brackets_fun<" << M << "," << N <<
```

```
        ",T>::FM_slice_brackets_fun(" << i << "): bad index";
      throw std::logic_error(err.str());
    }
  }

  int operator()(const FM_vector<M,T>& v, FM_vector<N,T>* res) const
  {
    *res = FM_vector<N,T>(static_cast<const T*>(v) + index);
    return FM_OK;
  }

private:
  const int index;
};


template <typename T>
class FM_swap_endian_fun : public std::unary_function<T,T>
{
public:
  int operator()(const T& t, T* res) const
  {
    union {
      T t;
      char chars[8];
    } u;
    char c;
    u.t = t;
    size_t sizeof_T = sizeof(T);
    switch(sizeof_T) {
    case 1:
      break;
    case 2:
      c = u.chars[0];
      u.chars[0] = u.chars[1];
      u.chars[1] = c;
      break;
    case 4:
      c = u.chars[0];
      u.chars[0] = u.chars[3];
      u.chars[3] = c;
      c = u.chars[1];
      u.chars[1] = u.chars[2];
      u.chars[2] = c;
      break;
    case 8:
      c = u.chars[0];
      u.chars[0] = u.chars[7];
      u.chars[7] = c;
      c = u.chars[1];
      u.chars[1] = u.chars[6];
      u.chars[6] = c;
      c = u.chars[2];
      u.chars[2] = u.chars[5];
      u.chars[5] = c;
      c = u.chars[3];
      u.chars[3] = u.chars[4];
      u.chars[4] = c;
      break;
    default:
      abort();
    }
    *res = u.t;
    return FM_OK;
  }
};


template <>
class FM_swap_endian_fun<int> : public std::unary_function<int,int>
{
public:
  int operator()(const int& i, int* res) const
  {
    *res =
      ((i & 0xFF) << 24) |
      ((i & 0xFF00) << 8) |
      ((i & 0xFF0000) >> 8) |
      ((i & 0xFF000000) >> 24);
    return FM_OK;
  }
};


template <>
class FM_swap_endian_fun<unsigned> :
  public std::unary_function<unsigned,unsigned>
{
public:
  int operator()(const unsigned& i, unsigned* res) const
  {
    *res =
      ((i & 0xFF) << 24) |
      ((i & 0xFF00) << 8) |
      ((i & 0xFF0000) >> 8) |
      ((i & 0xFF000000) >> 24);
    return FM_OK;
  }
};


template <>
class FM_swap_endian_fun<float> : public std::unary_function<float,float>
{
public:
  int operator()(const float& f, float* res) const
  {
    union {
      float f;
      int i;
    } u;
    u.f = f;
    u.i =
      ((u.i & 0xFF) << 24) |
      ((u.i & 0xFF00) << 8) |
      ((u.i & 0xFF0000) >> 8) |
      ((u.i & 0xFF000000) >> 24);
    *res = u.f;
    return FM_OK;
  }
};


template <typename T>
class FM_identity_fun : public std::unary_function<T,T>
{
public:
  int operator()(const T& t, T* res) const { *res = t; return FM_OK; }
};


template <typename T>
struct FM_first_greater_pred :  public std::binary_function<T,T,bool>
{
  bool operator()(const T& lhs, const T& rhs) const
  {
    return lhs.first > rhs.first;
  }
};


// T(S())
template <typename S, typename T>
class FM_compose_fun :
  public std::unary_function<typename S::argument_type,typename T::result_type>
{
public:
  FM_compose_fun(const S& s, const T& t) : first(s), second(t) {}

  typename T::result_type operator()(const typename S::argument_type& a) const
  {
    return second(first(a));
  }

private:
  const S first;
  const T second;
};

/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_INTERPOLATE_H_
#define _FM_INTERPOLATE_H_
/*
 * NAME: FM_interpolate.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <vector>
#include "FM_vector.h"
#include "FM_cell.h"
#include "FM_returns.h"


template <int B, typename T>
int FM_linear_interpolate(const FM_vector<B,FM_coord>& f,
                          T vals[], T* val)
{
  for (int i = B - 1; i >= 0; i--) {
    if (f[i] == FM_coord(0)) continue;
    int n = FM_pow_2(i);
    for (int j = 0; j < n; j++)
      vals[i] += f[i] * (vals[i + n] - vals[i]);
  }
  *val = vals[0];
  return FM_OK;
}


template <int B, typename T>
int
FM_linear_interpolate(const FM_base<B>& b,
                      const FM_ptr<FM_structured_B_cell<B> >& sc,
                      T vals[], T* val)
{
  FM_vector<B,FM_coord> f;
  for (int i = 0; i < B; i++)
    f[i] = b[i] - FM_coord(sc->get_index(i));
  return FM_linear_interpolate(f, vals, val);
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_IO_H_
#define _FM_IO_H_
/*
 * NAME: FM_io.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <vector>
#include <string>
#include <unistd.h>
#include <fcntl.h>
#include <assert.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include "FM_vector.h"
#include "FM_functional.h"


inline bool FM_little_endian_hardware()
{
  unsigned ui = 1;
  return static_cast<char*>(static_cast<void*>(&ui))[0] == 1 ? true : false;
}


size_t FM_file_size(const std::string& file_name)
{
  struct stat sbuf;
  int stat_res = stat(file_name.c_str(), &sbuf);
  return stat_res == 0 ? sbuf.st_size : 0;
}


template <typename T>
int FM_fread(FILE* fp, T* dat, size_t n_items, bool swap_endian)
{
  size_t fread_res = fread(dat, sizeof(T), n_items, fp);
  if (fread_res != n_items)
    return FM_IO_ERROR;
  if (swap_endian) {
    FM_swap_endian_fun<T> swap_endian_fun;
    for (size_t i = 0; i < fread_res; i++) {
      (void) swap_endian_fun(dat[i], &dat[i]);
    }
  }
  return FM_OK;
}


template <typename T>
int FM_fread(FILE* fp, T* dat, size_t n_items, bool swap_endian, bool fortran,
             std::vector<int>* extra = 0)
{
  int res;
  FM_u32 n_bytes_fortran;
  size_t n_bytes_expected = n_items * sizeof(T);
  size_t start = ftell(fp);
  if (fortran) {
    res = FM_fread(fp, &n_bytes_fortran, 1, swap_endian);
    if (res != FM_OK) return res;
    if (n_bytes_fortran < n_bytes_expected)
      return FM_IO_ERROR;
  }

  res = FM_fread(fp, dat, n_items, swap_endian);
  if (res != FM_OK) return res;

  if (fortran) {
    if (extra && n_bytes_fortran > n_bytes_expected) {
      size_t n_extra = (n_bytes_fortran - n_bytes_expected) / sizeof(T);
      extra->resize(n_extra);
      res = FM_fread(fp, &(*extra)[0], n_extra, swap_endian);
      if (res != FM_OK) return res;
    }
    (void) fseek(fp, start + n_bytes_fortran + 2 * sizeof(FM_u32), SEEK_SET);
  }

  return FM_OK;
}


template <typename F>
int FM_fread_fun(FILE* fp, typename F::result_type* dat, size_t n_items,
                 bool swap_endian, bool fortran, const F& fun)
{
  typedef typename F::argument_type argument_type;
  typedef typename F::result_type result_type;

  FM_u32 n_bytes_fortran;
  size_t start = ftell(fp);
  if (fortran) {
    FM_fread(fp, &n_bytes_fortran, 1, swap_endian);
    size_t n_bytes_expected = n_items * sizeof(argument_type);
    if (n_bytes_fortran < n_bytes_expected)
      return FM_IO_ERROR;
  }

  if (sizeof(argument_type) == sizeof(result_type)) {
    int res = FM_fread(fp, dat, n_items, swap_endian);
    if (res != FM_OK)
      return res;
    for (size_t i = 0; i < n_items; i++) {
      res = fun(reinterpret_cast<argument_type*>(dat)[i], &dat[i]);
      if (res != FM_OK)
        return res;
    }
  }
```

```
  else {
    std::vector<argument_type> tmp(n_items);
    int res = FM_fread(fp, &tmp[0], n_items, swap_endian);
    if (res != FM_OK)
      return res;
    for (size_t i = 0; i < n_items; i++) {
      res = fun(tmp[i], &dat[i]);
      if (res != FM_OK)
        return res;
    }
  }

  if (fortran)
    (void) fseek(fp, start + n_bytes_fortran + 2 * sizeof(FM_u32), SEEK_SET);

  return FM_OK;
}


template <int N, typename T>
int FM_fread_transpose(FILE* fps[], FM_vector<N,T>* dat,
                       size_t n_items, bool swap_endian, bool fortran)
{
  size_t start = ftell(fps[0]);
  size_t n_bytes_fortran;
  if (fortran) {
    FM_fread(fps[0], &n_bytes_fortran, 1, swap_endian);
    if (n_bytes_fortran < N * sizeof(T) * n_items)
      return FM_IO_ERROR;
  }

  size_t i, j;
  std::vector<T> components[N];
  T* bps[N];
  const FM_u32 N_DAT = 1024;
  for (i = 0; i < N; i++) {
    components[i].resize(N_DAT);
    size_t offset = start + i * sizeof(T) * n_items;
    if (fortran) offset += sizeof(FM_u32);
    (void) fseek(fps[i], offset, SEEK_SET);
  }

  size_t n_remaining = n_items;
  T* dst = reinterpret_cast<T*>(dat);
  while (n_remaining > 0) {
    size_t n_to_read = (n_remaining > N_DAT) ? N_DAT : n_remaining;
    for (i = 0; i < N; i++) {
      int res = FM_fread(fps[i], &components[i][0], n_to_read, swap_endian);
      if (res != FM_OK)
        return res;
      bps[i] = &components[i][0];
    }
    for (i = 0; i < n_to_read; i++)
      for (j = 0; j < N; j++)
        *dst++ = *bps[j]++;
    n_remaining -= n_to_read;
  }

  if (fortran)
    (void) fseek(fps[0], start + n_bytes_fortran + 2 * sizeof(FM_u32),
                 SEEK_SET);
  else
    (void) fseek(fps[0], ftell(fps[N - 1]), SEEK_SET);

  return FM_OK;
}


template <int N, typename F>
int FM_fread_transpose_fun(FILE* fps[],
                           FM_vector<N,typename F::result_type>* dat,
                           size_t n_items, bool swap_endian, bool fortran,
                           const F& fun)
{
  typedef typename F::argument_type argument_type;
  typedef typename F::result_type result_type;

  size_t start = ftell(fps[0]);
  size_t n_bytes_fortran;
  if (fortran) {
    FM_fread(fps[0], &n_bytes_fortran, 1, swap_endian);
    if (n_bytes_fortran < N * sizeof(argument_type) * n_items)
      return FM_IO_ERROR;
  }

  FM_u32 i, j;
  std::vector<argument_type> components[N];
  argument_type* bps[N];
  const FM_u32 N_DAT = 1024;
  for (i = 0; i < N; i++) {
    components[i].reserve(N_DAT);
    size_t offset = start + i * sizeof(argument_type) * n_items;
    if (fortran) offset += sizeof(FM_u32);
    (void) fseek(fps[i], offset, SEEK_SET);
  }

  size_t n_remaining = n_items;
  result_type* dst = reinterpret_cast<result_type*>(dat);
  int res;
  while (n_remaining > 0) {
    size_t n_to_read = (n_remaining > N_DAT) ? N_DAT : n_remaining;
    for (i = 0; i < N; i++) {
      res = FM_fread(fps[i], &components[i][0], n_to_read, swap_endian);
      if (res != FM_OK)
        return res;
      bps[i] = &components[i][0];
    }
    for (i = 0; i < n_to_read; i++) {
      res = fun(*bps[j]++, dst++);
      if (res != FM_OK)
        return res;
      }
    n_remaining -= n_to_read;
  }

  if (fortran)
    (void) fseek(fps[0], start + n_bytes_fortran + 2 * sizeof(FM_u32),
                 SEEK_SET);
  else
    (void) fseek(fps[0], ftell(fps[N - 1]), SEEK_SET);

  return FM_OK;
}


int FM_fskip(FILE* fp, size_t size, size_t n_items,
             bool swap_endian, bool fortran)
{
  FM_u32 n_bytes_fortran;
  size_t start = ftell(fp);
  if (fortran) {
    int res = FM_fread(fp, &n_bytes_fortran, 1, swap_endian);
    if (res != FM_OK)
      return res;
    size_t n_bytes_expected = n_items * size;
    if (n_bytes_fortran < n_bytes_expected)
      return FM_IO_ERROR;
    (void) fseek(fp, start + n_bytes_fortran + 2 * sizeof(FM_u32), SEEK_SET);
  }
  else {
    (void) fseek(fp, start + size * n_items, SEEK_SET);
  }
  return FM_OK;
}


template <typename T>
int FM_fwrite(FILE* fp, const T* dat, size_t n_items, bool swap_endian)
{
  size_t fwrite_res;
  if (!swap_endian) {
    fwrite_res = fwrite(dat, sizeof(T), n_items, fp);
  }
  else {
    std::vector<T> tmp(n_items);
    FM_swap_endian_fun<T> swap_endian_fun;
    for (FM_u32 i = 0; i < n_items; i++)
      (void) swap_endian_fun(dat[i], &tmp[i]);
    fwrite_res = fwrite(tmp, sizeof(T), n_items, fp);
  }
  return fwrite_res == n_items ? FM_OK : FM_IO_ERROR;
}


template <typename T>
size_t FM_fwrite(FILE* fp, const T* dat, size_t n_items,
                 bool swap_endian, bool fortran)
{
  int res;
  FM_u32 n_bytes_fortran = sizeof(T) * n_items;
  if (fortran) {
    res = FM_fwrite(fp, &n_bytes_fortran, sizeof(FM_u32), 1, swap_endian);
    if (res != FM_OK) return res;
  }
  res = FM_fwrite(fp, dat, n_items, swap_endian);
  if (res != FM_OK)
    return res;
  if (fortran) {
    res = FM_fwrite(fp, &n_bytes_fortran, sizeof(FM_u32), 1, swap_endian);
    if (res != FM_OK) return res;
  }
  return FM_OK;
}


template <int N, typename T>
int FM_fwrite_transpose(FILE* fp, const FM_vector<N,T>* dat,
                        FM_u32 n_items, bool swap_endian, bool fortran)
{
  int res;
  FM_u32 n_bytes_fortran = n_items * N * sizeof(T);
  if (fortran) {
    res = FM_write(fp, &n_bytes_fortran, 1, swap_endian);
    if (res != FM_OK) return res;
  }
  for (FM_u32 i = 0; i < N; i++) {
    for (FM_u32 j = 0; j < n_items; j++) {
      res = FM_fwrite(fp, &dat[j][i], 1, swap_endian);
      if (res != FM_OK) return res;
    }
  }
  if (fortran) {
    res = FM_write(fp, &n_bytes_fortran, 1, swap_endian);
    if (res != FM_OK) return res;
  }
  return FM_OK;
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
```

```
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */

#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_IRREGULAR_INTERVAL_H_
#define _FM_IRREGULAR_INTERVAL_H_
/*
 * NAME: FM_irregular_interval.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_structured_mesh.h"


class FM_irregular_interval : public FM_structured_mesh<1,1>
{
public:
  const FM_coord* const coordinates;
  const bool delete_supression;

  FM_irregular_interval(FM_u32 d, const FM_coord* c,
                        bool ds = false,
                        FM_properties_cache* pc = 0) :
    FM_structured_mesh<1,1>(d, pc), coordinates(c),
    delete_supression(ds)
  {
    for (FM_u32 i = 0; i < d - 1; i++) {
      if (!(c[i] < c[i + 1])) {
        FM_ostringstream err;
        err << "FM_irregular_interval::FM_irregular_interval: ";
        err << "coordinates must be strictly ascending";
        throw std::logic_error(err.str());
      }
    }
  }

  virtual ~FM_irregular_interval()
  {
    if (!delete_supression)
      delete [] coordinates;
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_irregular_interval";
  }

  virtual int
  at_cell(const FM_cell* c, std::vector<FM_vector<1,FM_coord> >* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[2];
    c->structured_mesh_vertex_indices(this, &n_indices, indices);
    for (FM_u32 i = 0; i < n_indices; i++)
      vals->push_back(coordinates[indices[i]]);
    return FM_OK;
  }

  virtual int
  at_cell(const FM_cell* c, FM_vector<1,FM_coord>* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[2];
    c->structured_mesh_vertex_indices(this, &n_indices, indices);
    for (FM_u32 i = 0; i < n_indices; i++)
      vals[i] = coordinates[indices[i]];
    return FM_OK;
  }

  virtual int
  phys_to_base(const FM_phys<1>& p, FM_context*, FM_base<1>* b,
               FM_ptr<FM_structured_B_cell<1> >* sc = 0) const
  {
    if (p[0] < coordinates[0] || p[0] > coordinates[dimensions[0] - 1])
      return FM_OUT_OF_BOUNDS;
    FM_u32 lo = 0, hi = dimensions[0] - 1;
    while (hi - lo > 1) {
      // assert(coordinates[lo] <= p[0] && p[0] <= coordinates[hi]
      int mid = (lo + hi) / 2;
      if (p[0] >= coordinates[mid])
        lo = mid;
      else
        hi = mid;
    }
    FM_u32 index = (lo < dimensions[0] - 1) ? lo : lo - 1;
    (*b)[0] = FM_coord(index) +
      (p[0] - coordinates[index]) /
      (coordinates[index + 1] - coordinates[index]);
    int res = FM_OK;
    if (sc)
      *sc = new FM_structured_B_cell<1>(index);
    return res;
  }

  virtual std::pair<FM_vector<1,FM_coord>,FM_vector<1,FM_coord> >
  get_bounding_box(const FM_time<FM_u32>* = 0, const FM_submesh_id* = 0) const
  {
    return std::pair<FM_vector<1,FM_coord>,FM_vector<1,FM_coord> >
      (coordinates[0], coordinates[dimensions[0] - 1]);
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
```

```
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_ITER_H_
#define _FM_ITER_H_
/*
 * NAME: FM_iter.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_shared_object.h"
#include "FM_cell.h"


enum FM_iter_attr_enum
{
  FM_ITER_ATTR_CELL_DIMENSION,
  FM_ITER_ATTR_CELL_TYPE,
  FM_ITER_ATTR_TIME,
  FM_ITER_ATTR_SUBMESH_ID,
  FM_ITER_ATTR_AXIS_BEGIN,
  FM_ITER_ATTR_AXIS_END,
  FM_ITER_ATTR_AXIS_STRIDE,
  FM_ITER_ATTR_SIMPLICIAL_DECOMPOSITION
};

class FM_iter_attr : public FM_shared_object
{
public:
  const FM_iter_attr_enum attr;

  FM_iter_attr(FM_iter_attr_enum a) : attr(a) {}
  virtual ~FM_iter_attr() {}
};


typedef std::vector<FM_ptr<FM_iter_attr> > FM_iter_attrs;

std::ostream& operator<<(std::ostream& o, const FM_iter_attrs& ia)
{
  o << "[";
  for (FM_u32 i = 0; i < ia.size(); i++) {
    if (i > 0)
      o << ", ";
    o << *ia[i];
  }
  return o << "]";
}


class FM_cell_dimension_iter_attr : public FM_iter_attr
{
public:
  const FM_u32 cell_dimension;
  FM_cell_dimension_iter_attr(FM_u32 cd) :
    FM_iter_attr(FM_ITER_ATTR_CELL_DIMENSION), cell_dimension(cd) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_cell_dimension_iter_attr(" << cell_dimension << ")";
  }
};

class FM_cell_type_iter_attr : public FM_iter_attr
{
public:
  const FM_cell_type_enum cell_type;
  FM_cell_type_iter_attr(FM_cell_type_enum ct) :
    FM_iter_attr(FM_ITER_ATTR_CELL_TYPE), cell_type(ct) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_cell_type_iter_attr(" << cell_type << ")";
  }
};

class FM_time_iter_attr : public FM_iter_attr
{
public:
  const FM_time<FM_u32> time;
  FM_time_iter_attr(const FM_time<FM_u32>& t) :
    FM_iter_attr(FM_ITER_ATTR_TIME), time(t) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_time_iter_attr(" << time << ")";
  }
};

class FM_submesh_id_iter_attr : public FM_iter_attr
{
public:
  const FM_submesh_id submesh_id;
  FM_submesh_id_iter_attr(const FM_submesh_id& sid) :
    FM_iter_attr(FM_ITER_ATTR_SUBMESH_ID), submesh_id(sid) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_submesh_id_iter_attr(" << submesh_id << ")";
  }
};

class FM_axis_begin_iter_attr : public FM_iter_attr
{
public:
  const FM_u32 axis, index;
  FM_axis_begin_iter_attr(FM_u32 a, FM_u32 i) :
    FM_iter_attr(FM_ITER_ATTR_AXIS_BEGIN), axis(a), index(i) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_axis_begin_iter_attr(" << axis << ", " << index << ")";
  }
};

class FM_axis_end_iter_attr : public FM_iter_attr
```

```
{
public:
  const FM_u32 axis, index;
  FM_axis_end_iter_attr(FM_u32 a, FM_u32 i) :
    FM_iter_attr(FM_ITER_ATTR_AXIS_END), axis(a), index(i) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_axis_end_iter_attr(" << axis << ", " << index << ")";
  }
};

class FM_axis_stride_iter_attr : public FM_iter_attr
{
public:
  const FM_u32 axis, stride;
  FM_axis_stride_iter_attr(FM_u32 a, FM_u32 s) :
    FM_iter_attr(FM_ITER_ATTR_AXIS_STRIDE), axis(a), stride(s) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_axis_stride_iter_attr(" << axis << ", " << stride << ")";
  }
};

class FM_simplicial_decomposition_iter_attr : public FM_iter_attr
{
public:
  const FM_u32 simplicial_decomposition;
  FM_simplicial_decomposition_iter_attr(FM_u32 sd) :
    FM_iter_attr(FM_ITER_ATTR_SIMPLICIAL_DECOMPOSITION),
    simplicial_decomposition(sd) {}
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_simplicial_decomposition_iter_attr(" <<
      simplicial_decomposition << ")";
  }
};


class FM_iter_impl
{
public:
  virtual ~FM_iter_impl() {}
  virtual FM_iter_impl* copy() const = 0;
  virtual const FM_cell* advance() = 0;
  virtual const FM_cell* dereference() const = 0;

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_iter_impl";
  }
};


class FM_iter
{
public:
  FM_iter() : impl(0), cell(0) {}
  FM_iter(FM_iter_impl* i) : impl(i), cell(impl->dereference()) {}
  FM_iter(const FM_iter& iter) :
    impl(iter.impl->copy()), cell(impl->dereference()) {}

  FM_iter& operator=(const FM_iter& rhs)
  {
    impl = rhs.impl->copy();
    cell = impl->dereference();
    return *this;
  }

  ~FM_iter()
  {
    if (impl) delete impl;
  }

  inline const FM_cell* operator++()
  {
    return cell = impl->advance();
  }
  void operator++(int) { (void) operator++(); }

  inline const FM_cell* operator*() const { return cell; }

  inline bool done() const { return cell == 0; }

  friend bool operator==(const FM_iter& lhs, const FM_iter& rhs)
  {
    if (lhs.cell == 0 || rhs.cell == 0)
      return lhs.cell == rhs.cell;
    else
      return *lhs.cell == *rhs.cell;
  }

  friend bool operator!=(const FM_iter& lhs, const FM_iter& rhs)
  {
    return !(lhs == rhs);
  }

private:
  FM_iter_impl* impl;
  const FM_cell* cell;
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_MATRIX_H_
#define _FM_MATRIX_H_
/*
 * NAME: FM_matrix.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_vector.h"


template <int M, int N, typename T>
FM_vector<N,T>
operator*(const FM_vector<M,T>& lhs,
          const FM_vector<M,FM_vector<N,T> >& rhs)
{
  T tmp[N], sum;
  for (int n = 0; n < N; n++) {
    sum = (T) 0;
    for (int m = 0; m < M; m++) {
      sum += lhs[m] * rhs[m][n];
    }
    tmp[n] = sum;
  }
  return FM_vector<N,T>(tmp);
}

template <typename T>
FM_vector<3,T>
operator*(const FM_vector<3,T>& lhs,
          const FM_vector<3,FM_vector<3,T> >& rhs)
{
  return FM_vector<3,T>(lhs[0] * rhs[0][0] +
                        lhs[1] * rhs[1][0] +
                        lhs[2] * rhs[2][0],
                        lhs[0] * rhs[0][1] +
                        lhs[1] * rhs[1][1] +
                        lhs[2] * rhs[2][1],
                        lhs[0] * rhs[0][2] +
                        lhs[1] * rhs[1][2] +
                        lhs[2] * rhs[2][2]);
}

template <typename T>
FM_vector<4,T>
operator*(const FM_vector<4,T>& lhs,
          const FM_vector<4,FM_vector<4,T> >& rhs)
{
  return FM_vector<4,T>(lhs[0] * rhs[0][0] +
                        lhs[1] * rhs[1][0] +
                        lhs[2] * rhs[2][0] +
                        lhs[3] * rhs[3][0],
                        lhs[0] * rhs[0][1] +
                        lhs[1] * rhs[1][1] +
                        lhs[2] * rhs[2][1] +
                        lhs[3] * rhs[3][1],
                        lhs[0] * rhs[0][2] +
                        lhs[1] * rhs[1][2] +
                        lhs[2] * rhs[2][2] +
                        lhs[3] * rhs[3][2],
                        lhs[0] * rhs[0][3] +
                        lhs[1] * rhs[1][3] +
                        lhs[2] * rhs[2][3] +
                        lhs[3] * rhs[3][3]);
}


template <int M, int N, typename T>
FM_vector<M,T>
operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
          const FM_vector<N,T>& rhs)
{
  T tmp[M];
  for (int m = 0; m < M; m++) {
    tmp[m] = FM_dot(lhs[m], rhs);
  }
  return FM_vector<M,T>(tmp);
}

template <typename T>
FM_vector<3,T>
operator*(const FM_vector<3,FM_vector<3,T> >& lhs,
          const FM_vector<3,T>& rhs)
{
  return FM_vector<3,T>(FM_dot(lhs[0], rhs),
                        FM_dot(lhs[1], rhs),
                        FM_dot(lhs[2], rhs));
}

template <typename T>
FM_vector<4,T>
operator*(const FM_vector<4,FM_vector<4,T> >& lhs,
          const FM_vector<4,T>& rhs)
{
  return FM_vector<4,T>(FM_dot(lhs[0], rhs),
                        FM_dot(lhs[1], rhs),
                        FM_dot(lhs[2], rhs),
                        FM_dot(lhs[3], rhs));
}


template <int M, int N, int P, typename T>
FM_vector<M,FM_vector<P,T> >
operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
          const FM_vector<N,FM_vector<P,T> >& rhs)
{
  FM_vector<P,T> tmp[M];
  T sum;
  for (int m = 0; m < M; m++) {
    for (int p = 0; p < P; p++) {
      sum = (T) 0;
      for (int n = 0; n < N; n++) {
        sum += lhs[m][n] * rhs[n][p];
      }
      tmp[m][p] = sum;
    }
  }
  return FM_vector<M,FM_vector<P,T> >(tmp);
}


template <int M, int N, typename T>
FM_vector<N,FM_vector<M,T> >
FM_transpose(const FM_vector<M,FM_vector<N,T> >& in)
{
  FM_vector<M,T> tmp[N];
  for (int m = 0; m < M; m++) {
    for (int n = 0; n < N; n++) {
      tmp[n][m] = in[m][n];
    }
  }
  return FM_vector<N,FM_vector<M,T> >(tmp);
}

template <typename T>
FM_vector<3,FM_vector<3,T> >
FM_transpose(const FM_vector<3,FM_vector<3,T> >& in)
{
  return FM_vector<3,FM_vector<3,T> >
    (FM_vector<3,T>(in[0][0], in[1][0], in[2][0]),
     FM_vector<3,T>(in[0][1], in[1][1], in[2][1]),
     FM_vector<3,T>(in[0][2], in[1][2], in[2][2]));
}


template <int N, typename T>
T FM_det(const FM_vector<N,FM_vector<N,T> >&);

template <int N, typename T>
T FM_minor(const FM_vector<N,FM_vector<N,T> >& in, int row, int col)
{
  FM_vector<N-1,FM_vector<N-1,T> > tmp;
  int dst_row, dst_col;
  dst_row = 0;
  for (int src_row = 0; src_row < N; src_row++) {
    if (src_row == row) continue;
    dst_col = 0;
    for (int src_col = 0; src_col < N; src_col++) {
      if (src_col == col) continue;
      tmp[dst_row][dst_col] = in[src_row][src_col];
      dst_col++;
    }
    dst_row++;
  }
  return FM_det(tmp);
}

template <int N, typename T>
T FM_det(const FM_vector<N,FM_vector<N,T> >& in)
{
  T sum = (T) 0;
  for (int n = 0; n < N; n++) {
    T minor = FM_minor(in, n, 0);
    T cofactor = (n & 1) ? -minor : minor;
    sum += in[n][0] * cofactor;
  }
  return sum;
}

template <typename T>
T FM_det(const FM_vector<1,FM_vector<1,T> >& in)
{
  return in[0][0];
}

template <typename T>
T FM_det(const FM_vector<2,FM_vector<2,T> >& in)
{
  return in[0][0] * in[1][1] - in[1][0] * in[0][1];
}

template <typename T>
T FM_det(const FM_vector<3,FM_vector<3,T> >& in)
{
  return
    in[0][0] * (in[1][1] * in[2][2] - in[2][1] * in[1][2]) -
    in[1][0] * (in[0][1] * in[2][2] - in[2][1] * in[0][2]) +
    in[2][0] * (in[0][1] * in[1][2] - in[1][1] * in[0][2]);
}

template <typename T>
T FM_det(const FM_vector<4,FM_vector<4,T> >& in)
{
  // columns 2,3
  T r0r1 = in[0][2] * in[1][3] - in[1][2] * in[0][3];
  T r0r2 = in[0][2] * in[2][3] - in[2][2] * in[0][3];
  T r0r3 = in[0][2] * in[3][3] - in[3][2] * in[0][3];
  T r1r2 = in[1][2] * in[2][3] - in[2][2] * in[1][3];
  T r1r3 = in[1][2] * in[3][3] - in[3][2] * in[1][3];
  T r2r3 = in[2][2] * in[3][3] - in[3][2] * in[2][3];

  // column 0
  T minor0 = in[1][1] * r2r3 - in[2][1] * r1r3 + in[3][1] * r1r2;
  T minor1 = in[0][1] * r2r3 - in[2][1] * r0r3 + in[3][1] * r0r2;
  T minor2 = in[0][1] * r1r3 - in[1][1] * r0r3 + in[3][1] * r0r1;
  T minor3 = in[0][1] * r1r2 - in[1][1] * r0r2 + in[2][1] * r0r1;

  return
    in[0][0] * minor0 -
```

```
    in[1][0] * minor1 +
    in[2][0] * minor2 -
    in[3][0] * minor3;
}


template <int N, typename T>
FM_vector<N,FM_vector<N,T> >
FM_adj(const FM_vector<N,FM_vector<N,T> >& in)
{
  FM_vector<N,FM_vector<N,T> > res;
  for (int row = 0; row < N; row++) {
    for (int col = 0; col < N; col++) {
      T minor = FM_minor(in, row, col);
      T cofactor = ((row + col) & 1) ? -minor : minor;
      res[col][row] = cofactor;  // transpose
    }
  }
  return res;
}


template <int N, typename T>
int FM_inv(const FM_vector<N,FM_vector<N,T> >& in,
           FM_vector<N,FM_vector<N,T> >* out)
{
  T det = FM_det(in);
  if (det == (T) 0)
    return 1;
  *out = (T) 1 / det * FM_adj(in);
  return 0;
}

template <typename T>
int FM_inv(const FM_vector<2,FM_vector<2,T> >& in,
           FM_vector<2,FM_vector<2,T> >* out)
{
  T det = FM_det(in);
  if (det == (T) 0)
    return 1;
  T inv_det = (T) 1 / det;
  (*out)[0][0] = inv_det * in[1][1];
  (*out)[0][1] = inv_det * -in[0][1];
  (*out)[1][0] = inv_det * -in[1][0];
  (*out)[1][1] = inv_det * in[0][0];
  return 0;
}

template <typename T>
int FM_inv(const FM_vector<3,FM_vector<3,T> >& in,
           FM_vector<3,FM_vector<3,T> >* out)
{
  // column 0
  T minor0 = in[1][1] * in[2][2] - in[2][1] * in[1][2];
  T minor1 = in[0][1] * in[2][2] - in[2][1] * in[0][2];
  T minor2 = in[0][1] * in[1][2] - in[1][1] * in[0][2];

  T det =
    in[0][0] * minor0 -
    in[1][0] * minor1 +
    in[2][0] * minor2;

  if (det == (T) 0)
    return 1;
  T inv_det = (T) 1 / det;

  (*out)[0][0] = inv_det * minor0;
  (*out)[0][1] = inv_det * -minor1;
  (*out)[0][2] = inv_det * minor2;
  (*out)[1][0] = inv_det * (in[2][0] * in[1][2] - in[1][0] * in[2][2]);
  (*out)[1][1] = inv_det * (in[0][0] * in[2][2] - in[2][0] * in[0][2]);
  (*out)[1][2] = inv_det * (in[1][0] * in[0][2] - in[0][0] * in[1][2]);
  (*out)[2][0] = inv_det * (in[1][0] * in[2][1] - in[2][0] * in[1][1]);
  (*out)[2][1] = inv_det * (in[2][0] * in[0][1] - in[0][0] * in[2][1]);
  (*out)[2][2] = inv_det * (in[0][0] * in[1][1] - in[1][0] * in[0][1]);

  return 0;
}


template <typename T>
int FM_inv(const FM_vector<4,FM_vector<4,T> >& in,
           FM_vector<4,FM_vector<4,T> >* out)
{
  // compute minors column by column, but fill in (*out) row
  // by row to effectively transpose

  // columns 2,3
  T r0r1 = in[0][2] * in[1][3] - in[1][2] * in[0][3];
  T r0r2 = in[0][2] * in[2][3] - in[2][2] * in[0][3];
  T r0r3 = in[0][2] * in[3][3] - in[3][2] * in[0][3];
  T r1r2 = in[1][2] * in[2][3] - in[2][2] * in[1][3];
  T r1r3 = in[1][2] * in[3][3] - in[3][2] * in[1][3];
  T r2r3 = in[2][2] * in[3][3] - in[3][2] * in[2][3];

  // column 0
  T minor0 = in[1][1] * r2r3 - in[2][1] * r1r3 + in[3][1] * r1r2;
  T minor1 = in[0][1] * r2r3 - in[2][1] * r0r3 + in[3][1] * r0r2;
  T minor2 = in[0][1] * r1r3 - in[1][1] * r0r3 + in[3][1] * r0r1;
  T minor3 = in[0][1] * r1r2 - in[1][1] * r0r2 + in[2][1] * r0r1;

  T det =
    in[0][0] * minor0 -
    in[1][0] * minor1 +
    in[2][0] * minor2 -
    in[3][0] * minor3;

  if (det == (T) 0)
    return 1;
  T inv_det = (T) 1 / det;

  (*out)[0][0] = inv_det * minor0;
  (*out)[0][1] = inv_det * -minor1;
  (*out)[0][2] = inv_det * minor2;
  (*out)[0][3] = inv_det * -minor3;

  // column 1
  minor0 = in[1][0] * r2r3 - in[2][0] * r1r3 + in[3][0] * r1r2;
  minor1 = in[0][0] * r2r3 - in[2][0] * r0r3 + in[3][0] * r0r2;
  minor2 = in[0][0] * r1r3 - in[1][0] * r0r3 + in[3][0] * r0r1;
  minor3 = in[0][0] * r1r2 - in[1][0] * r0r2 + in[2][0] * r0r1;

  (*out)[1][0] = inv_det * -minor0;
  (*out)[1][1] = inv_det * minor1;
  (*out)[1][2] = inv_det * -minor2;
  (*out)[1][3] = inv_det * minor3;

  // columns 0,1
  r0r1 = in[0][0] * in[1][1] - in[1][0] * in[0][1];
  r0r2 = in[0][0] * in[2][1] - in[2][0] * in[0][1];
  r0r3 = in[0][0] * in[3][1] - in[3][0] * in[0][1];
  r1r2 = in[1][0] * in[2][1] - in[2][0] * in[1][1];
  r1r3 = in[1][0] * in[3][1] - in[3][0] * in[1][1];
  r2r3 = in[2][0] * in[3][1] - in[3][0] * in[2][1];

  // column 2
  minor0 = in[1][3] * r2r3 - in[2][3] * r1r3 + in[3][3] * r1r2;
  minor1 = in[0][3] * r2r3 - in[2][3] * r0r3 + in[3][3] * r0r2;
  minor2 = in[0][3] * r1r3 - in[1][3] * r0r3 + in[3][3] * r0r1;
  minor3 = in[0][3] * r1r2 - in[1][3] * r0r2 + in[2][3] * r0r1;

  (*out)[2][0] = inv_det * minor0;
  (*out)[2][1] = inv_det * -minor1;
  (*out)[2][2] = inv_det * minor2;
  (*out)[2][3] = inv_det * -minor3;

  // column 3
  minor0 = in[1][2] * r2r3 - in[2][2] * r1r3 + in[3][2] * r1r2;
  minor1 = in[0][2] * r2r3 - in[2][2] * r0r3 + in[3][2] * r0r2;
  minor2 = in[0][2] * r1r3 - in[1][2] * r0r3 + in[3][2] * r0r1;
  minor3 = in[0][2] * r1r2 - in[1][2] * r0r2 + in[2][2] * r0r1;

  (*out)[3][0] = inv_det * -minor0;
  (*out)[3][1] = inv_det * minor1;
  (*out)[3][2] = inv_det * -minor2;
  (*out)[3][3] = inv_det * minor3;

  return 0;
}


template <int N, typename T>
void FM_identity(FM_vector<N,FM_vector<N,T> >* out)
{
  T zero = (T) 0;
  T one = (T) 1;
  for (int row = 0; row < N; row++)
    for (int col = 0; col < N; col++)
      (*out)[row][col] = row == col ? one : zero;
}


typedef FM_vector<2,FM_vector<2,float> > FM_matrix22f;
typedef FM_vector<2,FM_vector<3,float> > FM_matrix23f;
typedef FM_vector<3,FM_vector<2,float> > FM_matrix32f;
typedef FM_vector<3,FM_vector<3,float> > FM_matrix33f;
typedef FM_vector<3,FM_vector<3,double> > FM_matrix33d;
typedef FM_vector<4,FM_vector<4,float> > FM_matrix44f;
typedef FM_vector<4,FM_vector<4,double> > FM_matrix44d;


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_MESH_H_
#define _FM_MESH_H_
/*
 * NAME: FM_mesh.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_shared_object_with_properties_cache.h"
#include "FM_field_interface.h"
#include "FM_iter.h"
#include "FM_interpolate.h"


class FM_mesh_ : public FM_shared_object_with_properties_cache
{
public:
  const FM_u32 base_dimensionality;
  const FM_u32 phys_dimensionality;

  FM_mesh_(FM_u32 b, FM_u32 d, FM_properties_cache* pc) :
    FM_shared_object_with_properties_cache(pc),
    base_dimensionality(b),
    phys_dimensionality(d)
  {}

  virtual FM_u64 get_card(FM_u32, FM_u32 = 0, const FM_time<FM_u32>* = 0,
                          const FM_submesh_id* = 0) const = 0;

  virtual bool get_structured_behavior(const FM_time<FM_u32>* = 0,
                                       const FM_submesh_id* = 0) const
  {
    return false;
  }

  virtual FM_u64 cell_to_enum(const FM_cell*) const = 0;
  virtual FM_ptr<FM_cell> enum_to_cell(FM_u64, FM_u32, FM_u32 = 0,
                                       const FM_time<FM_u32>* = 0,
                                       const FM_submesh_id* = 0) const = 0;

  virtual std::vector<FM_ptr<FM_cell> >
  faces(const FM_cell*, FM_u32) const = 0;

  virtual std::vector<FM_ptr<FM_cell> >
  adjacencies(const FM_cell* c) const
  {
    std::vector<FM_ptr<FM_cell> > adjacent_cells;
    FM_u32 d = c->get_dimension();
    if (d == 0 || d == base_dimensionality) {
      FM_u32 d1 = (d == 0 ? 1 : d - 1);
      std::vector<FM_ptr<FM_cell> > cells1 = faces(c, d1);
      std::vector<FM_ptr<FM_cell> >::const_iterator iter;
      for (iter = cells1.begin(); iter != cells1.end(); ++iter) {
        std::vector<FM_ptr<FM_cell> > cells = faces(*iter, d);
        if (cells.size() == size_t(2))
          adjacent_cells.push_back(*cells[0] != *c ? cells[0] : cells[1]);
      }
      return adjacent_cells;
    }

    FM_ostringstream err;
    err << *this << "::adjacencies(" << *c << "): " <<
      "only for cells with dimension 0 or B (" << base_dimensionality << ")";
    throw std::logic_error(err.str());
  }

  virtual FM_iter begin() const = 0;
  virtual FM_iter begin(const FM_iter_attrs&) const = 0;
  inline FM_iter end() const { return FM_iter(); }

  virtual FM_ptr<FM_shared_object>
  get_aux(const std::string& key, FM_u32 pass,
          const FM_time<FM_u32>* t, const FM_submesh_id* sid) const
  {
    if (key == "base_dimensionality") {
      return new FM_simple_value<FM_u32>(base_dimensionality);
    }
    else if (key == "phys_dimensionality") {
      return new FM_simple_value<FM_u32>(phys_dimensionality);
    }
    else if (key == "n_submeshes") {
      return new FM_simple_value<FM_u32>(0);
    }
    else if (key == "structured_behavior") {
      return new FM_simple_value<bool>(get_structured_behavior(t, sid));
    }
    return FM_shared_object_with_properties_cache::get_aux(key, pass, t, sid);
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& property_names,
                         const FM_time<FM_u32>* t,
                         const FM_submesh_id* sid) const
  {
    property_names.insert("base_dimensionality");
    property_names.insert("phys_dimensionality");
    property_names.insert("n_submeshes");
    property_names.insert("structured_behavior");
    return FM_shared_object_with_properties_cache::
      get_property_names_aux(property_names, t, sid);
  }
};


template <int B, int D>
class FM_mesh :
  public FM_mesh_,
  public FM_field_interface<B,D,FM_vector<D,FM_coord> >
{
public:
  FM_mesh(FM_properties_cache* pc) :
    FM_mesh_(FM_u32(B), FM_u32(D), pc),
    FM_field_interface<B,D,FM_vector<D,FM_coord> >(this, 0)
  {}

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_mesh<" << B << "," << D << ">";
  }

  virtual FM_vector<B,FM_u32>
  get_base_dimensions(const FM_time<FM_u32>* t = 0,
                      const FM_submesh_id* sid = 0) const
  {
    FM_ostringstream err;
    err << *this << "::get_base_dimensions(" <<
      (t ? *t : FM_time<FM_u32>()) << ", " <<
      (sid ? *sid : FM_submesh_id()) << "): not for this mesh";
    throw std::logic_error(err.str());
  }

  virtual std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> >
  get_bounding_box(const FM_time<FM_u32>* t = 0,
                   const FM_submesh_id* sid = 0) const
  {
    std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> > bb;
    FM_iter_attrs iter_attrs;
    if (t && t->defined())
      iter_attrs.push_back(new FM_time_iter_attr(*t));
    if (sid && sid->defined())
      iter_attrs.push_back(new FM_submesh_id_iter_attr(*sid));
    FM_iter iter = begin(iter_attrs);
    FM_iter e = end();
    FM_vector<D,FM_coord> cv;
    int res = FM_OK;
    bool first = true;
    for ( ; iter != e; ++iter) {
      res = at_cell(*iter, &cv);
      if (res != FM_OK) break;
      if (first) {
        bb.first = cv;
        bb.second = cv;
        first = false;
        continue;
      }
      FM_operator_min_max_equals(bb, cv);
    }
    if (res != FM_OK) {
      FM_ostringstream err;
      err << *this << "::get_bounding_box(" <<
        (t ? *t : FM_time<FM_u32>()) << ", " <<
        (sid ? *sid : FM_submesh_id()) << "): got res " << res;
      throw std::logic_error(err.str());
    }
    return bb;
  }

  virtual int
  at_phys(const FM_phys<D>& p, FM_context* ctxt,
          FM_vector<D,FM_coord>* val) const
  {
    FM_ptr<FM_cell> c;
    int res = phys_to_cell(p, ctxt, &c);
    if (res != FM_OK) return res;
    *val = p;
    return res;
  }

  virtual int
  base_to_cell(const FM_base<B>&, FM_ptr<FM_structured_B_cell<B> >*) const
  {
    return FM_NOT_DEFINED;
  }

  virtual int
  base_to_phys(const FM_base<B>& b, FM_context* ctxt, FM_phys<D>* p) const
  {
    int res = at_base(b, ctxt, p);
    p->time.set_undefined();
    return res;
  }

  virtual int
  phys_to_base(const FM_phys<D>&, FM_context*, FM_base<B>*,
               FM_ptr<FM_structured_B_cell<B> >* = 0) const
  {
    return FM_NOT_DEFINED;
  }

  virtual int
  phys_to_cell(const FM_phys<D>&, FM_context*, FM_ptr<FM_cell>*) const = 0;


  virtual FM_vector<D,FM_coord> centroid(const FM_cell* c) const
  {
    std::vector<FM_vector<D,FM_coord> > vertex_coordinates;
    int res = at_cell(c, &vertex_coordinates);
    if (res != FM_OK) {
      FM_ostringstream err;
      err << *this << "::centroid(" << *c << "): at_cell res " << res;
      throw std::logic_error(err.str());
    }
    FM_vector<D,FM_coord> cen = vertex_coordinates[0];
    for (size_t i = 1; i < vertex_coordinates.size(); i++)
      cen += vertex_coordinates[i];
    cen *= FM_coord(1) / FM_coord(vertex_coordinates.size());
    return cen;
  }
```

```
  virtual FM_ptr<FM_shared_object>
  get_aux(const std::string& key, FM_u32 pass,
          const FM_time<FM_u32>* t, const FM_submesh_id* sid) const
  {
    if (key == "bounding_box") {
      std::pair<FM_vector<D,FM_coord>, FM_vector<D,FM_coord> > min_max =
        get_bounding_box(t, sid);
      std::vector<FM_ptr<FM_shared_object> > lo_values(D);
      std::vector<FM_ptr<FM_shared_object> > hi_values(D);
      for (int j = 0; j < D; j++) {
        lo_values[j] = new FM_simple_value<FM_coord>(min_max.first[j]);
        hi_values[j] = new FM_simple_value<FM_coord>(min_max.second[j]);
      }
      return new FM_tuple_value(new FM_tuple_value(lo_values),
                                new FM_tuple_value(hi_values));
    }
    return FM_mesh_::get_aux(key, pass, t, sid);
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& property_names,
                         const FM_time<FM_u32>* t,
                         const FM_submesh_id* sid) const
  {
    property_names.insert("bounding_box");
    return FM_mesh_::get_property_names_aux(property_names, t, sid);
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_MUTEX_H_
#define _FM_MUTEX_H_
/*
 * NAME: FM_mutex.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <pthread.h>


class FM_mutex
{
public:
  FM_mutex() { pthread_mutex_init(&mutex, 0); }
  ~FM_mutex() { pthread_mutex_destroy(&mutex); }

  inline int lock() { return pthread_mutex_lock(&mutex); }
  inline int unlock() { return pthread_mutex_unlock(&mutex); }

private:
  pthread_mutex_t mutex;
};

/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_ORIENT_H_
#define _FM_ORIENT_H_
/*
 * NAME: FM_orient.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <stdlib.h>
#include <assert.h>
#include "FM_vector.h"
#include "FM_combinatorics.h"


const char* FM_orientation_names[3] = {
  "outside", "orientation 0", "inside"
};


// Return:
//    1 if c is to the left of the line through ab (from a to b)
//   -1 if c is to the right of the line through ab
//    0 if c is colinear with ab
//
// This is equivalent to evaluating the sign of the determinant:
//
//    | a[0] a[1] 1 |
//    | b[0] b[1] 1 |
//    | c[0] c[1] 1 |
//
template <typename T>
int FM_orient(const FM_vector<2,T>& a,
              const FM_vector<2,T>& b,
              const FM_vector<2,T>& c)
{
  return FM_sign(a[0] * (b[1] - c[1]) - b[0] * (a[1] - c[1]) +
                 c[0] * (a[1] - b[1]));
}

// Return:
//    1 if d sees triangle abc vertices counterclockwise
//   -1 if d sees triangle abc vertices clockwise
//    0 if d is coplanar with abc
//
template <typename T>
int FM_orient(const FM_vector<3,T>& a, const FM_vector<3,T>& b,
              const FM_vector<3,T>& c, const FM_vector<3,T>& d)
{
  return FM_sign(FM_dot(d - a, FM_cross(b - a, c - a)));
}


template <>
int FM_orient(const FM_vector<3,float>& a, const FM_vector<3,float>& b,
              const FM_vector<3,float>& c, const FM_vector<3,float>& d)
{
  // computed manually for performance, doubles for fewer round-off problems
  double ba_x = b[0] - a[0];
  double ba_y = b[1] - a[1];
  double ba_z = b[2] - a[2];
  double ca_x = c[0] - a[0];
  double ca_y = c[1] - a[1];
  double ca_z = c[2] - a[2];
  return FM_sign((d[0] - a[0]) * (ba_y * ca_z - ca_y * ba_z) +
                 (d[1] - a[1]) * (ca_x * ba_z - ba_x * ca_z) +
                 (d[2] - a[2]) * (ba_x * ca_y - ca_x * ba_y));
}


// Compute orientation of e with respect to quadrilateral abcd
// by treating abcd as two triangles: abc and acd.
//
// Let there be a viewer who sees abcd counter-clockwise (e.g. from
// a position inside a 3-cell).  Return:
//
//    1 if e on same side of quadrilateral abcd as viewer
//   -1 if e on opposite side of quadrilateral abcd with respect to viewer
//    0 if e is coplanar with abcd
//
// The result is computed by doing two orientation tests, one for e
// with respect to abc, and one for e with respect to acd.
// If these two tests agree, we conclude we're done.  If they don't,
// take the largest magnitude result, based on triangles abd, abc,
// bcd, and acd.
//
template <typename T>
int FM_orient(const FM_vector<3,T>& a, const FM_vector<3,T>& b,
              const FM_vector<3,T>& c, const FM_vector<3,T>& d,
              const FM_vector<3,T>& e)
{
  FM_vector<3,double> ba(b - a);
  FM_vector<3,double> ca(c - a);
  FM_vector<3,double> da(d - a);
  FM_vector<3,double> ea(e - a);

  int abc_orient = FM_sign(FM_dot(ea, FM_cross(ba, ca)));
  int acd_orient = FM_sign(FM_dot(ea, FM_cross(ca, da)));

  if (abc_orient == acd_orient)
    return abc_orient;

  int verbosity = 1;
  if (verbosity > 0) {
    std::cerr << "FM_orient(" << a << ", " << b << ", " << c << ", " <<
      d << ", " << e << ")" << std::endl;
    std::cerr << "abc_orient: " << abc_orient;
    std::cerr << ", acd_orient: " << acd_orient << std::endl;
  }

  FM_vector<3,double> cb(c - b);
  FM_vector<3,double> dc(d - c);
  FM_vector<3,double> eb(e - b);
  FM_vector<3,double> ec(e - c);
  FM_vector<3,double> ed(e - d);

  const char* triangle_names[4] = {"abd", "abc", "bcd", "acd"};
  double results[4];
  results[0] = FM_dot(ea, FM_cross(ba, da));
  results[1] = FM_dot(eb, FM_cross(cb, -ba));
  results[2] = FM_dot(ec, FM_cross(dc, -cb));
  results[3] = FM_dot(ed, FM_cross(-da, -dc));

  int i, largest_result = 0;
  for (i = 1; i < 4; i++) {
    if (FM_abs(results[i]) > FM_abs(results[largest_result]))
      largest_result = i;
  }

  int res = FM_sign(results[largest_result]);

  if (verbosity > 0) {
    for (i = 0; i < 4; i++)
      std::cerr << triangle_names[i] << " orient result: " <<
        results[i] << std::endl;

    std::cerr << "largest magnitude result " <<
      "(triangle " << triangle_names[largest_result] << "): " <<
      results[largest_result] << std::endl;

    std::cerr << "FM_orient(" << a << ", " << b << ", " << c << ", " <<
      d << ", " << e << ") returning " << res << std::endl;
  }

  return res;
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_OSTRINGSTREAM_H_
#define _FM_OSTRINGSTREAM_H_
/*
 * NAME: FM_ostringstream.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 *
 * DESCRIPTION:
 *      FM_ostringstream provides a work-around for systems
 *      that do not have std::ostringstream yet, using the
 *      deprecated std::ostrstream.  We basically define a
 *      new class (FM_ostringstream) derived from std::strstream,
 *      with a redefined str() method that handles string termination
 *      and memory management properly.
 *
 * SEE ALSO:
 *      N. Josuttis.  The C++ Standard Library:  A Tutorial and
 *      Reference, Addison-Wesley, 2000, pages 649-651.  This
 *      is the best reference I have found that explains how
 *      strstream works.
 */
#if defined(__GNUC__)
#ifndef FM_NO_STRINGSTREAM
#define FM_NO_STRINGSTREAM
#endif
#endif


#ifndef FM_NO_STRINGSTREAM
#include <sstream>
typedef std::ostringstream FM_ostringstream;
#else
#include <strstream>
#include <string>

class FM_ostringstream : public std::ostrstream
{
public:
  std::string str()
  {
    // the std::ostrstream::str() method does not automatically
    // terminate the string with 0, so we must ensure that it is
    // terminated ourselves.
    *this << std::ends;

    // The std::ostrstream::str() call internally calls freeze()
    // on the buffer, meaning that ownership of the memory is
    // transferred to the caller.  We do not want to be responsible
    // for the deallocation (for among other reasons because we do
    // not know how it was allocated) so we "unfreeze" to transfer
    // deallocation duties back to ostrstream.
    const char* res = std::ostrstream::str();
    freeze(false);
    return std::string(res);
  }
};

#endif


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_PHYS_H_
#define _FM_PHYS_H_
/*
 * NAME: FM_phys.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_vector.h"
#include "FM_time.h"

template <int D>
class FM_phys : public FM_vector<D,FM_coord>
{
public:
  FM_time<FM_coord> time;

  FM_phys() {}
  FM_phys(const FM_coord dat[]) : FM_vector<D,FM_coord>(dat) {}
  FM_phys(const FM_vector<D,FM_coord>& v) : FM_vector<D,FM_coord>(v) {}
};


template <>
class FM_phys<1> : public FM_vector<1,FM_coord>
{
public:
  FM_time<FM_coord> time;

  FM_phys() {}
  FM_phys(const FM_coord dat[]) : FM_vector<1,FM_coord>(dat) {}
  FM_phys(const FM_vector<1,FM_coord>& v) : FM_vector<1,FM_coord>(v) {}
  FM_phys(const FM_coord& a0) :
    FM_vector<1,FM_coord>(a0) {}
};


template <>
class FM_phys<2> : public FM_vector<2,FM_coord>
{
public:
  FM_time<FM_coord> time;

  FM_phys() {}
  FM_phys(const FM_coord dat[]) : FM_vector<2,FM_coord>(dat) {}
  FM_phys(const FM_vector<2,FM_coord>& v) : FM_vector<2,FM_coord>(v) {}
  FM_phys(const FM_coord& a0, const FM_coord& a1) :
    FM_vector<2,FM_coord>(a0, a1) {}
};


template <>
class FM_phys<3> : public FM_vector<3,FM_coord>
{
public:
  FM_time<FM_coord> time;

  FM_phys() {}
  FM_phys(const FM_coord dat[]) : FM_vector<3,FM_coord>(dat) {}
  FM_phys(const FM_vector<3,FM_coord>& v) : FM_vector<3,FM_coord>(v) {}
  FM_phys(const FM_coord& a0, const FM_coord& a1, const FM_coord& a2) :
    FM_vector<3,FM_coord>(a0, a1, a2) {}
};


template <int D>
std::ostream& operator<<(std::ostream& lhs, const FM_phys<D>& rhs)
{
  lhs << "(";
  int i;
  for (i = 0; i < D; i++) {
    if (i > 0) lhs << ", ";
    lhs << rhs[i];
  }
  if (rhs.time.defined()) {
    if (i++ > 0) lhs << ", ";
    lhs << "time=" << rhs.time.get();
  }
  return lhs << ")";
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

```
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_PRODUCT_MESH_H_
#define _FM_PRODUCT_MESH_H_
/*
 * NAME: FM_product_mesh.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_structured_mesh.h"


template <int B, int D>
class FM_product_mesh : public FM_structured_mesh<B,D>
{
public:
  const std::vector<FM_ptr<FM_structured_mesh<1,1> > > axes;

  FM_product_mesh(const std::vector<FM_ptr<FM_structured_mesh<1,1> > >& a,
                  FM_properties_cache* pc = 0) :
    FM_structured_mesh<B,D>(pc),
    axes(a)
  {
    if (axes.size() != size_t(B)) {
      FM_ostringstream err;
      err << "FM_product_mesh<" << B << "," << D << ">::FM_product_mesh: ";
      err << "expecting " << B << " axes, got " << axes.size();
      throw std::logic_error(err.str());
    }

    FM_vector<1,FM_u32> dimension;
    for (int i = 0; i < B; i++) {
      dimension = axes[i]->get_base_dimensions();
      const_cast<FM_u32&>(dimensions[i]) = dimension[0];
    }
    init(dimensions);
  }

protected:
  FM_product_mesh(const FM_vector<B,FM_u32>& d, FM_properties_cache* pc = 0) :
    FM_structured_mesh<B,D>(d, pc),
    axes(B)
  {
    // axes filled in by derived class constructor
  }

public:
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_product_mesh<" << B << "," << D << ">";
  }

  virtual int
  at_cell(const FM_cell* c, std::vector<FM_vector<D,FM_coord> >* vals) const
  {
    FM_u32 d = c->get_dimension();
    FM_u32 n = c->is_subsimplex() ? d + 1 : FM_pow_2(d);
    FM_u32 previous_size = vals->size();
    vals->resize(previous_size + n);
    return FM_product_mesh::at_cell(c, &(*vals)[previous_size]);
  }

  virtual int
  at_cell(const FM_cell* c, FM_vector<D,FM_coord>* vals) const
  {
    const FM_structured_cell<B>* sc =
      dynamic_cast<const FM_structured_cell<B>*>(c);
    if (sc == 0)
      FM_throw_bad_cell_argument(this, "at_cell", c);

    FM_base<1,FM_u32> b(c->get_time(), c->get_submesh_id());
    FM_u32 d = sc->get_dimension();
    FM_vector<1,FM_coord> coordinate;
    if (sc->is_subsimplex()) {
      //assert(B == 2 || B == 3);
      FM_u32 subid = sc->get_subid();
      FM_u32 n = d + 1;
      for (FM_u32 i = 0; i < n; i++) {
        FM_u32 mask = 1;
        int j;
        for (j = 0; j < B; j++) {
          b[0] = sc->get_index(j);
          if (FM_structured_hexahedron_subfaces[d][subid][i] & mask)
            b[0]++;
          int res = axes[j]->at_base(b, &coordinate);
          if (res != FM_OK) return res;
          vals[i][j] = coordinate[0];
          mask <<= 1;
        }
        for ( ; j < D; j++)
          vals[i][j] = FM_coord(0);
      }
    }
    else {
      FM_u32 n = FM_pow_2(d);
      FM_vector<B,bool> free_indices = alignments[d][sc->get_alignment()];
      for (FM_u32 i = 0; i < n; i++) {
        FM_u32 mask = 1;
        int j;
        for (j = 0; j < B; j++) {
          b[0] = sc->get_index(j);
          if (free_indices[j] && (i & mask))
            b[0]++;
          int res = axes[j]->at_base(b, &coordinate);
          if (res != FM_OK) return res;
          vals[i][j] = coordinate[0];
          mask <<= 1;
        }
        for ( ; j < D; j++)
          vals[i][j] = FM_coord(0);
```

```
      }
    }
    return FM_OK;
  }

  virtual int phys_to_base(const FM_phys<D>& p, FM_context* ctxt,
                           FM_base<B>* b,
                           FM_ptr<FM_structured_B_cell<B> >* sc = 0) const
  {
    int res;
    FM_base<1> b1;
    FM_vector<B,FM_u32> indices;
    for (int i = 0; i < B; i++) {
      res = axes[i]->phys_to_base(p[i], ctxt, &b1);
      if (res != FM_OK) return res;
      (*b)[i] = b1[0];
      indices[i] = FM_u32((*b)[i]);
      if (indices[i] == dimensions[i] - 1) indices[i]--;
    }
    if (sc)
      *sc = new FM_structured_B_cell<B>(indices);
    return res;
  }

  virtual std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> >
  get_bounding_box(const FM_time<FM_u32>* t = 0,
                   const FM_submesh_id* sid = 0) const
  {
    std::pair<FM_vector<D,FM_coord>,FM_vector<D,FM_coord> > bb;
    int i;
    for (i = 0; i < B; i++) {
      std::pair<FM_vector<1,FM_coord>,FM_vector<1,FM_coord> >
        axis_bounding_interval = axes[i]->get_bounding_box(t, sid);
      bb.first[i] = axis_bounding_interval.first[0];
      bb.second[i] = axis_bounding_interval.second[0];
    }
    for ( ; i < D; i++) {
      bb.first[i] = FM_coord(0);
      bb.second[i] = FM_coord(0);
    }
    return bb;
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_PROPERTIES_CACHE_H_
#define _FM_PROPERTIES_CACHE_H_
/*
 * NAME: FM_properties_cache.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <map>
#include "FM_shared_object.h"


typedef std::string FM_properties_key;
typedef std::map<FM_properties_key,FM_ptr<FM_shared_object> >
FM_properties_map;

class FM_properties_cache : public FM_shared_object
{
public:
  FM_properties_cache() {}
  FM_properties_cache(const FM_properties_map& props) :
    properties_cache(props) {}

  virtual FM_ptr<FM_shared_object>
  get(const std::string& key,
      const FM_time<FM_u32>* = 0, const FM_submesh_id* = 0) const
  {
    FM_ptr<FM_shared_object> res;
    properties_cache_mutex.lock();
    FM_properties_map::const_iterator
      properties_iter = properties_cache.find(key);
    if (properties_iter != properties_cache.end())
      res = (*properties_iter).second;
    properties_cache_mutex.unlock();
    return res;
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& property_names,
                         const FM_time<FM_u32>*, const FM_submesh_id*) const
  {
    properties_cache_mutex.lock();
    FM_properties_map::const_iterator i;
    for (i = properties_cache.begin(); i != properties_cache.end(); ++i)
      property_names.insert((*i).first);
    properties_cache_mutex.unlock();
    return property_names;
  }

  virtual void
  set(const std::string& key, const FM_shared_object* property,
      const FM_time<FM_u32>* = 0, const FM_submesh_id* = 0)
  {
    properties_cache_mutex.lock();
    properties_cache[key] = property;
    properties_cache_mutex.unlock();
  }

private:
  FM_properties_map properties_cache;
  mutable FM_mutex properties_cache_mutex;
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_REGULAR_INTERVAL_H_
#define _FM_REGULAR_INTERVAL_H_
/*
 * NAME: FM_regular_interval.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_structured_mesh.h"

class FM_regular_interval : public FM_structured_mesh<1,1>
{
public:
  const FM_coord origin;
  const FM_coord spacing;

  FM_regular_interval(FM_u32 d,
                      FM_coord o = FM_coord(0),
                      FM_coord s = FM_coord(1),
                      FM_properties_cache* pc = 0) :
    FM_structured_mesh<1,1>(d, pc), origin(o), spacing(s) {}

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_regular_interval";
  }

  virtual int
  at_cell(const FM_cell* c, std::vector<FM_vector<1,FM_coord> >* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[2];
    c->structured_mesh_vertex_indices(this, &n_indices, indices);
    for (FM_u32 i = 0; i < n_indices; i++)
      vals->push_back(origin + FM_coord(indices[i]) * spacing);
    return FM_OK;
  }

  virtual int
  at_cell(const FM_cell* c, FM_vector<1,FM_coord>* vals) const
  {
    FM_u32 n_indices;
    FM_u64 indices[2];
    c->structured_mesh_vertex_indices(this, &n_indices, indices);
    for (FM_u32 i = 0; i < n_indices; i++)
      vals[i] = origin + FM_coord(indices[i]) * spacing;
    return FM_OK;
  }

  virtual int
  phys_to_base(const FM_phys<1>& p, FM_context*, FM_base<1>* b,
               FM_ptr<FM_structured_B_cell<1> >* sc = 0) const
  {
    (*b)[0] = (p[0] - origin) / spacing;
    int res = (FM_coord(0) <= (*b)[0] &&
               (*b)[0] <= FM_coord(dimensions[0] - 1)) ?
      FM_OK : FM_OUT_OF_BOUNDS;
    if (res != FM_OK) return res;
    if (sc)
      res = base_to_cell(*b, sc);
    return res;
  }

  virtual std::pair<FM_vector<1,FM_coord>,FM_vector<1,FM_coord> >
  get_bounding_box(const FM_time<FM_u32>* = 0, const FM_submesh_id* = 0) const
  {
    return std::pair<FM_vector<1,FM_coord>,FM_vector<1,FM_coord> >
      (origin, origin + FM_coord(dimensions[0] - 1) * spacing);
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_REGULAR_MESH_H_
#define _FM_REGULAR_MESH_H_
/*
 * NAME: FM_regular_mesh.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_product_mesh.h"
#include "FM_regular_interval.h"


template <int B, int D>
class FM_regular_mesh : public FM_product_mesh<B,D>
{
public:
  FM_regular_mesh(const FM_vector<B,FM_u32>& d, FM_properties_cache* pc = 0) :
    FM_product_mesh<B,D>(d, pc)
  {
    for (int i = 0; i < B; i++)
      const_cast<FM_ptr<FM_structured_mesh<1,1> >&>(axes[i]) =
        new FM_regular_interval(d[i]);
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_regular_mesh<" << B << "," << D << ">";
  }
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_RESULTS_H_
#define _FM_RESULTS_H_
/*
 * NAME: FM_results.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */

const int FM_OK = 0;
const int FM_OUT_OF_BOUNDS = 1;
const int FM_BLANKED_DATA = 2;
const int FM_POINT_LOCATION_FAILED = 3;
const int FM_IO_ERROR = 4;
const int FM_INTERPOLATION_ERROR = 5;
const int FM_NOT_DEFINED = 6;
const int FM_POINT_LOCATE_WALKED_OFF_MESH = 7;
const int FM_POINT_LOCATE_STUCK = 8;
const int FM_POINT_OUTSIDE_BOUNDING_BOX = 9;

/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_SHARED_OBJECT_H_
#define _FM_SHARED_OBJECT_H_
/*
 * NAME: FM_shared_object.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <iostream>
#include <vector>
#include <set>
#include <stdexcept>
#include <typeinfo>
#include <assert.h>
#include "FM_mutex.h"
#include "FM_submesh_id.h"
#include "FM_time.h"
#include "FM_returns.h"
#include "FM_ostringstream.h"


// FM_ptr is a "smart pointer" for pointing at FM_shared_object's.
template <typename T>
class FM_ptr
{
public:
  FM_ptr() : ptr(0) {}
  FM_ptr(const T* p) : ptr(p)
  {
    if (ptr) ptr->increment_reference();
  }

  // NOTE:
  // It is essential to still provide the copy constructor
  // and assignment operator definitions as untemplated member
  // functions, in addition to the corresponding templated
  // member functions.  Without the untemplated member functions,
  // the compiler will silently generate its default code
  // when it encounters an argument of the type *this.  The
  // implicit code does not manage reference counts properly,
  // so we must prevent the compiler from emitting it.
  //

  template <typename S>
  FM_ptr(const FM_ptr<S>& p) :
    ptr(dynamic_cast<const T*>(static_cast<const S*>(p)))
  {
    if (ptr) {
      ptr->increment_reference();
    }
    else {
      if (static_cast<const S*>(p)) {
        FM_ostringstream err;
        err << "FM_ptr<" << typeid(T).name() << ">::FM_ptr(FM_ptr<" <<
          typeid(S).name() << ">&): bad dynamic cast";
        throw std::logic_error(err.str());
      }
    }
  }

  FM_ptr(FM_ptr<T>& p) : ptr(static_cast<const T*>(p))
  {
    if (ptr) ptr->increment_reference();
  }

  template <typename S>
  const FM_ptr<T>& operator=(const FM_ptr<S>& rhs)
  {
    const T* tmp = dynamic_cast<const T*>(static_cast<const S*>(rhs));
    if (static_cast<const S*>(rhs) && !tmp) {
      FM_ostringstream err;
      err << "const FM_ptr<" << typeid(T).name() <<
        ">& operator=(const FM_ptr<" <<
        typeid(S).name() << ">&): bad dynamic cast";
      throw std::logic_error(err.str());
    }
    set(tmp);
    return *this;
  }

  const FM_ptr<T>& operator=(const FM_ptr<T>& rhs)
  {
    set(static_cast<const T*>(rhs));
    return *this;
  }

  ~FM_ptr()
  {
    if (ptr) {
      if (ptr->decrement_reference() == 0) {
        delete ptr;
      }
    }
  }

  inline const T* operator->() const { return ptr; }
  inline operator const T*() const { return ptr; }

  void set(const T* t)
  {
    if (t) t->increment_reference();
    if (ptr) {
      if (ptr->decrement_reference() == 0) {
        delete ptr;
      }
    }
    ptr = t;
  }
```

```
  const FM_ptr<T>& operator=(const T* rhs)
  {
    set(rhs);
    return *this;
  }

protected:
  const T* ptr;
};


template <typename S, typename T>
bool operator==(const FM_ptr<S>& lhs, const FM_ptr<T>& rhs)
{
  return static_cast<const S*>(lhs) == static_cast<const T*>(rhs);
}

// We can get the following from some STL implementations, to
// avoid ambiguity problems we do not define our own if we get
// STL's definition:
//
//template <class _Tp>
//inline bool operator!=(const _Tp& __x, const _Tp& __y) {
//  return !(__x == __y);
//}
#ifndef __SGI_STL_INTERNAL_RELOPS
template <typename S, typename T>
bool operator!=(const FM_ptr<S>& lhs, const FM_ptr<T>& rhs)
{
  return !(lhs == rhs);
}
#endif

template <typename T>
std::ostream& operator<<(std::ostream& s, const FM_ptr<T>& p)
{
  return s << static_cast<const T*>(p);
}

// FM_shared_object is the base class for reference counted shared objects.
class FM_shared_object
{
public:
  FM_shared_object() : reference_count(0)
  {
#ifdef FM_SHARED_OBJECT_DEBUG
    // do not call virtual functions (re)defined by subclasses yet ...
    // std::cout << *this << " constructing" << std::endl;
    std::cout << this << " constructing" << std::endl;
#endif
  }

  virtual ~FM_shared_object()
  {
#ifdef FM_SHARED_OBJECT_DEBUG
    // do not call virtual functions (re)defined by subclasses now ...
    // std::cout << *this << " destructing" << std::endl;
    std::cout << this << " destructing" << std::endl;
#endif
    assert(reference_count == 0);
  }

  inline int get_reference_count() const { return reference_count; }

  virtual std::ostream& str(std::ostream& o) const
  {
    return o << this;
  }

  friend std::ostream& operator<<(std::ostream& o, const FM_shared_object& so)
  {
    return so.str(o);
  }

  virtual FM_ptr<FM_shared_object>
  get(const std::string& key,
      const FM_time<FM_u32>* t = 0, const FM_submesh_id* sid = 0) const
  {
    FM_ptr<FM_shared_object> property = get_aux(key, 0, t, sid);
    if (property == 0)
      property = get_aux(key, 1, t, sid);
    if (!property) {
      FM_ostringstream err;
      err << *this << "::get(\"" << key << ", ";
      err << (t ? *t : FM_time<FM_u32>()) << ", ";
      err << (sid ? *sid : FM_submesh_id()) << "\"): undefined";
      throw std::logic_error(err.str());
    }
    return property;
  }

  virtual std::set<std::string>
  get_property_names(const FM_time<FM_u32>* t = 0,
                     const FM_submesh_id* sid = 0) const
  {
    std::set<std::string> property_names;
    return get_property_names_aux(property_names, t, sid);
  }

  template <typename T>
  void get_simple_value(const std::string&, T*, const FM_time<FM_u32>* = 0,
                        const FM_submesh_id* = 0) const;

protected:
  virtual FM_ptr<FM_shared_object>
  get_aux(const std::string&, FM_u32,
          const FM_time<FM_u32>*, const FM_submesh_id*) const
  {
    return 0;
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& property_names,
                         const FM_time<FM_u32>*, const FM_submesh_id*) const
  {
    return property_names;
  }

private:
  template <typename T> friend class FM_ptr;

  int increment_reference() const
  {
    mutex.lock();
    int res = ++reference_count;
#ifdef FM_SHARED_OBJECT_DEBUG
    std::cout << *this << " increment_reference(), now "
              << reference_count << std::endl;
#endif
    mutex.unlock();
    return res;
  }

  int decrement_reference() const
  {
    mutex.lock();
    int res = --reference_count;
#ifdef FM_SHARED_OBJECT_DEBUG
    std::cout << *this << " decrement_reference(), now "
              << reference_count << std::endl;
#endif
    mutex.unlock();
    assert (res >= 0);
    return res;
  }

// prevent copy construction and assignment
private:
  FM_shared_object(const FM_shared_object&);
  const FM_shared_object& operator=(const FM_shared_object&);

private:
  mutable int reference_count;
  mutable FM_mutex mutex;
};


template <typename T>
class FM_simple_value : public FM_shared_object
{
public:
  const T value;

  FM_simple_value(const T& v) : value(v) {}
  virtual std::ostream& str(std::ostream& o) const { return o << value; }
};


class FM_tuple_value : public FM_shared_object
{
public:
  FM_tuple_value(const std::vector<FM_ptr<FM_shared_object> >& vs) :
    values(vs) {}

  FM_tuple_value(FM_shared_object* v0) : values(1)
  {
    values[0] = v0;
  }

  FM_tuple_value(FM_shared_object* v0, FM_shared_object* v1) :
    values(2)
  {
    values[0] = v0;
    values[1] = v1;
  }

  inline size_t size() const { return values.size(); }
  const FM_ptr<FM_shared_object>& operator[](int i) const
  {
    return values[i];
  }

  virtual std::ostream& str(std::ostream& o) const
  {
    o << "(";
    for (size_t i = 0; i < values.size(); i++) {
      if (i > 0) o << ", ";
      o << *values[i];
    }
    return o << ")";
  }

private:
  std::vector<FM_ptr<FM_shared_object> > values;
};


template <typename T>
void FM_shared_object::
get_simple_value(const std::string& key, T* v, const FM_time<FM_u32>* t,
                 const FM_submesh_id* sid) const
{
  FM_ptr<FM_shared_object> so_sv = get(key, t, sid);
  const FM_simple_value<T>& sv =
    dynamic_cast<const FM_simple_value<T>&>(*so_sv);
  *v = sv.value;
}


/*
```

```
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_SHARED_OBJECT_WITH_PROPERTIES_CACHE_H_
#define _FM_SHARED_OBJECT_WITH_PROPERTIES_CACHE_H_
/*
 * NAME: FM_shared_object_with_properties_cache.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_shared_object.h"
#include "FM_properties_cache.h"


class FM_shared_object_with_properties_cache : public FM_shared_object
{
public:
  FM_shared_object_with_properties_cache() {}
  FM_shared_object_with_properties_cache(FM_properties_cache* pc) :
    properties_cache(pc ? pc : new FM_properties_cache()) {}

  virtual FM_ptr<FM_shared_object>
  get(const std::string& key,
      const FM_time<FM_u32>* t = 0, const FM_submesh_id* sid = 0) const
  {
    // 1. check cache
    FM_ptr<FM_shared_object> property = properties_cache->get(key, t, sid);
    if (property)
      return property;

    // 2. first pass: bottom up, through class lineage
    property = get_aux(key, 0, t, sid);

    // 3. second pass: opportunity to query composed classes
    property = get_aux(key, 1, t, sid);

    if (!property) {
      FM_ostringstream err;
      err << *this << "::get(\"" << key << "\", ";
      err << ((t) ? *t : FM_time<FM_u32>()) << ", ";
      err << ((sid) ? *sid : FM_submesh_id()) << "): property undefined";
      throw std::logic_error(err.str());
    }

    // do not cache if property specific to a time step or submesh
    if (!(((t) && t->defined()) || ((sid) && sid->defined()))) {
      const_cast<FM_properties_cache*>
        (static_cast<const FM_properties_cache*>(properties_cache))->
        set(key, property, t, sid);
    }

    return property;
  }

  virtual void set(const std::string& key, const FM_shared_object* property,
                   const FM_time<FM_u32>* t = 0, const FM_submesh_id* sid = 0)
  {
    const_cast<FM_properties_cache*>
      (static_cast<const FM_properties_cache*>(properties_cache))->
      set(key, property, t, sid);
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& property_names,
                         const FM_time<FM_u32>* t,
                         const FM_submesh_id* sid) const
  {
    property_names =
      properties_cache->get_property_names_aux(property_names, t, sid);
    return FM_shared_object::get_property_names_aux(property_names, t, sid);
  }

protected:
  FM_ptr<FM_properties_cache> properties_cache;
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_STRUCTURED_MESH_H_
#define _FM_STRUCTURED_MESH_H_
/*
 * NAME: FM_structured_mesh.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_mesh.h"
#include "FM_combinatorics.h"
#include "FM_orient.h"

//
// The canonical vertex numbering for a structured hexahedron.
// The vertex indices in the tables below are in terms of this
// hexahedron numbering.
//
//                 6_______7
//                /|       /|  ^
//               / |      / |  |
//             4/_______5/  |  k
//             |  2|___ |___|3
//             |  /     |  /  ^
//             | /      | /  /
//             |/_______|/  j
//             0        1
//                i ->
//
// Hexahedron faces:
FM_u32 FM_hf0_0[] = {0};
FM_u32 FM_hsf0_0[] = {0};
FM_u32* FM_hf0[] = {FM_hf0_0};
FM_u32* FM_hsf0[] = {FM_hsf0_0};

FM_u32 FM_hf1_0[] = {0, 1};
FM_u32 FM_hf1_1[] = {0, 2};
FM_u32 FM_hf1_2[] = {0, 4};
FM_u32 FM_hsf1_0[] = {0, 3};
FM_u32 FM_hsf1_1[] = {0, 5};
FM_u32 FM_hsf1_2[] = {0, 6};
FM_u32 FM_hsf1_3[] = {1, 2};
FM_u32 FM_hsf1_4[] = {1, 4};
FM_u32 FM_hsf1_5[] = {2, 4};
FM_u32* FM_hf1[] = {
  FM_hf1_0, FM_hf1_1, FM_hf1_2
};
FM_u32* FM_hsf1[] = {
  FM_hsf1_0, FM_hsf1_1, FM_hsf1_2, FM_hsf1_3, FM_hsf1_4, FM_hsf1_5
};

FM_u32 FM_hf2_0[] = {0, 1, 2, 3};
FM_u32 FM_hf2_1[] = {0, 1, 4, 5};
FM_u32 FM_hf2_2[] = {0, 2, 4, 6};

FM_u32 FM_hsf2_0[] = {0, 1, 3};         //              2___3   2___3
FM_u32 FM_hsf2_1[] = {0, 3, 2};         // K-surface    |1 /|   |\ 3|
FM_u32 FM_hsf2_2[] = {0, 1, 2};         //              |/_0|   |2 \|
FM_u32 FM_hsf2_3[] = {1, 3, 2};         //              0   1   0   1

FM_u32 FM_hsf2_4[] = {0, 5, 1};         //              4___5   4___5
FM_u32 FM_hsf2_5[] = {0, 4, 5};         // J-surface    |5 /|   |\ 7|
FM_u32 FM_hsf2_6[] = {0, 4, 1};         //              |/_4|   |6_\|
FM_u32 FM_hsf2_7[] = {1, 4, 5};         //              0   1   0   1

FM_u32 FM_hsf2_8[] = {0, 2, 6};         //              4___6   4___6
FM_u32 FM_hsf2_9[] = {0, 6, 4};         // I-surface    |9 /|   |\11|
FM_u32 FM_hsf2_10[] = {0, 2, 4};        //              |/_8|   |10\|
FM_u32 FM_hsf2_11[] = {2, 6, 4};        //              0   2   0   2

// triangles interior to hexahedron

FM_u32 FM_hsf2_12[] = {0, 5, 3};        // even 5-tetrahedra decomposition
FM_u32 FM_hsf2_13[] = {0, 3, 6};
FM_u32 FM_hsf2_14[] = {0, 6, 5};
FM_u32 FM_hsf2_15[] = {3, 5, 6};

FM_u32 FM_hsf2_16[] = {1, 2, 4};        // odd 5-tetrahedra decomposition
FM_u32 FM_hsf2_17[] = {1, 4, 7};
FM_u32 FM_hsf2_18[] = {1, 7, 2};
FM_u32 FM_hsf2_19[] = {2, 7, 4};

FM_u32* FM_hf2[] = {
  FM_hf2_0, FM_hf2_1, FM_hf2_2
};
FM_u32* FM_hsf2[] = {
  FM_hsf2_0, FM_hsf2_1, FM_hsf2_2, FM_hsf2_3, FM_hsf2_4, FM_hsf2_5, FM_hsf2_6,
  FM_hsf2_7, FM_hsf2_8, FM_hsf2_9, FM_hsf2_10, FM_hsf2_11, FM_hsf2_12,
  FM_hsf2_13, FM_hsf2_14, FM_hsf2_15, FM_hsf2_16, FM_hsf2_17, FM_hsf2_18,
  FM_hsf2_19
};

FM_u32 FM_hf3_0[] = {0, 1, 2, 3, 4, 5, 6, 7};
FM_u32 FM_hsf3_0[] = {3, 6, 5, 7};
FM_u32 FM_hsf3_1[] = {0, 3, 5, 1};
FM_u32 FM_hsf3_2[] = {0, 6, 3, 2};
FM_u32 FM_hsf3_3[] = {0, 5, 6, 4};
FM_u32 FM_hsf3_4[] = {0, 5, 3, 6};
FM_u32 FM_hsf3_5[] = {1, 4, 2, 0};
FM_u32 FM_hsf3_6[] = {1, 2, 7, 3};
FM_u32 FM_hsf3_7[] = {1, 7, 4, 5};
FM_u32 FM_hsf3_8[] = {2, 4, 7, 6};
FM_u32 FM_hsf3_9[] = {1, 2, 4, 7};
FM_u32* FM_hf3[] = {FM_hf3_0};
FM_u32* FM_hsf3[] = {
  FM_hsf3_0, FM_hsf3_1, FM_hsf3_2, FM_hsf3_3, FM_hsf3_4,
  FM_hsf3_5, FM_hsf3_6, FM_hsf3_7, FM_hsf3_8, FM_hsf3_9
};

FM_u32** FM_structured_hexahedron_faces[] = {
  FM_hf0, FM_hf1, FM_hf2, FM_hf3
};

FM_u32** FM_structured_hexahedron_subfaces[4] = {
  FM_hsf0, FM_hsf1, FM_hsf2, FM_hsf3
};

const FM_u32 FM_subtetrahedron_face[10][4][3] = {
  // even hexahedron cases
  {{3, 6, 5}, {3, 5, 7}, {3, 7, 6}, {5, 6, 7}},
  {{0, 3, 5}, {0, 1, 3}, {0, 5, 1}, {1, 5, 3}},
  {{0, 6, 3}, {0, 3, 2}, {0, 2, 6}, {2, 3, 6}},
  {{0, 5, 6}, {0, 4, 5}, {0, 6, 4}, {4, 6, 5}},
  {{0, 5, 3}, {0, 3, 6}, {0, 6, 5}, {3, 5, 6}},

  // odd hexahedron cases
  {{1, 4, 2}, {0, 1, 2}, {0, 2, 4}, {0, 4, 1}},
  {{1, 2, 7}, {1, 3, 2}, {1, 7, 3}, {2, 3, 7}},
  {{1, 7, 4}, {1, 4, 5}, {1, 5, 7}, {4, 7, 5}},
  {{2, 4, 7}, {2, 6, 4}, {2, 7, 6}, {4, 6, 7}},
  {{1, 2, 4}, {1, 4, 7}, {1, 7, 2}, {2, 7, 4}}
};

const FM_u32 FM_hexahedron_face[2][6][4] = {
  // even hexahedron case
  {{0, 2, 6, 4}, {3, 1, 5, 7}, {0, 4, 5, 1},
   {3, 7, 6, 2}, {0, 1, 3, 2}, {5, 4, 6, 7}},

  // odd hexahedron cases
  {{2, 6, 4, 0}, {1, 5, 7, 3}, {1, 0, 4, 5},
   {2, 3, 7, 6}, {1, 3, 2, 0}, {4, 6, 7, 5}}
};

struct FM_cell_step
{
  int dir;
  FM_u32 axis;
  FM_u32 subsimplex;
  FM_u32 subsimplex_face;
};

const FM_cell_step FM_tetrahedron_step[10][4] = {
  // even hexahedron cases
  { // stepping from subtetrahedron 0
    { 0, 0, 4, 3},
    { 1, 0, 8, 1},
    { 1, 1, 7, 1},
    { 1, 2, 6, 1}
  },
  { // stepping from subtetrahedron 1
    { 0, 0, 4, 0},
    {-1, 2, 7, 3},
    {-1, 1, 6, 3},
    { 1, 0, 5, 2}
  },
  { // stepping from subtetrahedron 2
    { 0, 0, 4, 1},
    {-1, 2, 8, 3},
    {-1, 0, 6, 2},
    { 1, 1, 5, 3}
  },
  { // stepping from subtetrahedron 3
    { 0, 0, 4, 2},
    {-1, 1, 8, 2},
    {-1, 0, 7, 2},
    { 1, 2, 5, 1}
  },
  { // stepping from subtetrahedron 4
    { 0, 0, 1, 0},
    { 0, 0, 2, 0},
    { 0, 0, 3, 0},
    { 0, 0, 0, 0}
  },

  // odd hexahedron cases
  { // stepping from subtetrahedron 5
    { 0, 0, 9, 0},
    {-1, 2, 3, 3},
    {-1, 0, 1, 3},
    {-1, 1, 2, 3}
  },
  { // stepping from subtetrahedron 6
    { 0, 0, 9, 2},
    {-1, 2, 0, 3},
    { 1, 0, 2, 2},
    { 1, 1, 1, 2}
  },
  { // stepping from subtetrahedron 7
    { 0, 0, 9, 1},
    {-1, 1, 0, 2},
    { 1, 0, 3, 2},
    { 1, 2, 1, 1}
  },
  { // stepping from subtetrahedron 8
    { 0, 0, 9, 3},
    {-1, 0, 0, 1},
    { 1, 1, 3, 1},
    { 1, 2, 2, 1}
  },
  { // stepping from subtetrahedron 9
    { 0, 0, 5, 0},
    { 0, 0, 7, 0},
    { 0, 0, 6, 0},
    { 0, 0, 8, 0}
  }
};


//
// Helper routines for FM_structured_mesh<B,D>:  Most routines
```

```
// assist topological member functions such as faces() and adjacencies().
// These functions in general are dependent upon the base dimensionality B,
// but not the physical_dimensionality D, thus they are templated only
// by B rather than by both B and D.
//

template <int B>
bool FM_simplicial_decomposition_odd(const FM_vector<B,FM_u32>& indices,
                                     FM_u32 simplicial_decomposition)
{
  bool res = FM_odd(indices);
  if (simplicial_decomposition == 2) res = !res;
  return res;
}


template <int B, int D>
class FM_structured_mesh;


template <int B, int D>
void FM_structured_mesh_init(FM_structured_mesh<B,D>* sm,
                             const FM_vector<B,FM_u32>& d)
{
  const_cast<FM_vector<B,FM_u32>&>(sm->dimensions) = d;

  int i, j;
  for (i = 0; i < B; i++) {
    if (!(sm->dimensions[i] > 1)) {
      FM_ostringstream err;
      err << "FM_structured_mesh<" << B << ">";
      err << "::init(" << sm->dimensions << "): ";
      err << "each dimension must be > 1";
      throw std::logic_error(err.str());
    }
  }

  for (i = 0; i <= B; i++) {
    FM_choose_choices<B>(FM_u32(i), &sm->alignments[i]);
    std::vector<FM_vector<B,bool> >::const_iterator iter;
    for (iter = sm->alignments[i].begin(); iter != sm->alignments[i].end();
         ++iter) {
      FM_u64 card = 1;
      FM_vector<B,FM_u32> dims;
      for (j = 0; j < B; j++) {
        dims[j] = (*iter)[j] ? sm->dimensions[j] - 1 : sm->dimensions[j];
        card *= dims[j];
      }
      sm->alignment_dimensions[i].push_back(dims);
      const_cast<std::vector<FM_u64>&>(sm->alignment_cards[i]).push_back(card);
    }
  }
  const_cast<FM_u64&>(sm->cube_offsets[0]) = 0;
  FM_u64 offset = 1;
  for (i = 0; i < B; i++) {
    int n = FM_pow_2(i);
    for (j = 0; j < n; j++)
      const_cast<FM_u64&>(sm->cube_offsets[j + n]) =
        sm->cube_offsets[j] + offset;
    offset*= sm->dimensions[i];
  }

  /*
    std::cout << "FM_structured_mesh<" << B << ">("
              << dimensions << "):" << std::endl;
    for (i = 0; i <= B; i++) {
      std::cout << "alignments[" << i << "]:";
      for (j = 0; j < alignments[i].size(); j++)
        std::cout << " " << alignments[i][j];
      std::cout << std::endl;

      std::cout << "alignment_dimensions[" << i << "]:";
      for (j = 0; j < alignments[i].size(); j++)
        std::cout << " " << alignment_dimensions[i][j];
      std::cout << std::endl;

      std::cout << "alignment_cards[" << i << "]:";
      for (j = 0; j < alignments[i].size(); j++)
        std::cout << " " << alignment_cards[i][j];
      std::cout << std::endl;
    }

    std::cout << "cube_offsets:";
    for (i = 0; i < 1 << B; i++)
      std::cout << " " << cube_offsets[i];
    std::cout << std::endl;
  */
}

FM_u64 FM_structured_mesh_get_card(FM_u32 base_dimensionality,
                                   const std::vector<FM_u64> alignment_cards[],
                                   FM_u32 k)
{
  if (k > base_dimensionality) return 0;
  FM_u64 card = 0;
  for (size_t j = 0; j < alignment_cards[k].size(); j++)
    card += alignment_cards[k][j];
  return card;
}

template <int B>
FM_u64 FM_structured_mesh_cell_to_enum
(const std::vector<FM_vector<B,FM_u32> > alignment_dimensions[],
 const std::vector<FM_u64> alignment_cards[],
 const FM_cell* c)
{
  const FM_structured_cell<B>* sc =
    dynamic_cast<const FM_structured_cell<B>*>(c);
  FM_u32 d = sc->get_dimension();
  FM_u32 a = sc->get_alignment();
  FM_u64 index = sc->get_index(B - 1);
  for (int i = B - 2; i >= 0; i--)
    index = index * alignment_dimensions[d][a][i] + sc->get_index(i);
  for (FM_u32 j = 0; j < a; j++)
    index += alignment_cards[d][j];
  return index;
}

template <int B>
FM_ptr<FM_cell> FM_structured_mesh_enum_to_cell
(const std::vector<FM_vector<B,bool> > alignments[],
 const std::vector<FM_vector<B,FM_u32> > alignment_dimensions[],
 const std::vector<FM_u64> alignment_cards[],
 FM_u64 e, FM_u32 d, FM_u32,
 const FM_time<FM_u32>* t, const FM_submesh_id* sid)
{
  size_t a;
  FM_u64 e_original = e;
  for (a = 0; a < alignments[d].size(); a++) {
    if (e < alignment_cards[d][a]) break;
    e -= alignment_cards[d][a];
  }
  if (a == alignments[d].size()) {
    FM_ostringstream err;
    err << "FM_structured_mesh<" << B << ",D>::enum_to_cell(" <<
      e_original << ", " << d << "): ";
    err << "bad argument";
    throw std::logic_error(err.str());
  }
  FM_vector<B,FM_u64> strides;
  int i;
  strides[0] = 1;
  for (i = 1; i < B; i++)
    strides[i] = strides[i - 1] * alignment_dimensions[d][a][i - 1];
  FM_vector<B,FM_u32> indices;
  for (i = B - 1; i > 0; i--) {
    div_t d = div(e, strides[i]);
    indices[i] = d.quot;
    e = d.rem;
  }
  indices[0] = e;

  FM_cell* res = new FM_structured_k_cell<B>(d, a, indices);
  if (t)
    res->set_time(*t);
  if (sid)
    res->set_submesh_id(*sid);
  return res;
}

template <int B>
std::vector<FM_ptr<FM_cell> >
FM_structured_mesh_faces(const FM_vector<B,FM_u32>& dimensions,
                         const std::vector<FM_vector<B,bool> > alignments[],
                         const FM_cell* c, FM_u32 k)
{
  const FM_structured_cell<B>* sc =
    dynamic_cast<const FM_structured_cell<B>*>(c);
  assert(sc != 0);

  FM_u32 d = sc->get_dimension();
  FM_u32 a = sc->get_alignment();

  std::vector<FM_ptr<FM_cell> > cells;

  // g++ version 2.96 20000731 (Red Hat Linux 7.1 2.96-81)
  // crashes when compiling -O3, do not use ref here:
  // const FM_vector<B,bool>& alignment = alignments[d][a];
  const FM_vector<B,bool> alignment = alignments[d][a];

  for (a = 0; a < alignments[k].size(); a++) {
    if (k > d) {
      if (!(alignment <= alignments[k][a]))
        continue;
    }
    else {
      if (!(alignment >= alignments[k][a]))
        continue;
    }
    FM_vector<B,bool> free_indices = alignment ^ alignments[k][a];
    FM_u32 n_candidates = FM_pow_2(k > d ? k - d : d - k);
    for (FM_u32 i = 0; i < n_candidates; i++) {
      bool out_of_bounds = false;
      FM_vector<B,FM_u32> indices = sc->get_indices();
      FM_u32 free_index_mask = 1;
      for (int j = 0; j < B; j++) {
        if (free_indices[j]) {
          if (k > d) {
            if (i & free_index_mask) {
              out_of_bounds = indices[j] == 0;
              if (!out_of_bounds)
                --indices[j];
            }
            else
              out_of_bounds = indices[j] >= dimensions[j] - 1;
            if (out_of_bounds) break;
          }
          else {
            if (i & free_index_mask)
              ++indices[j];
          }
          free_index_mask <<= 1;
        }
      }
      if (out_of_bounds) continue;
      cells.push_back(new FM_structured_k_cell<B>(c->get_time(),
                                                  c->get_submesh_id(),
                                                  k, a, indices));
    }
```

```
  }
  return cells;
}

template <int B>
std::vector<FM_ptr<FM_cell> >
FM_structured_mesh_adjacencies(const FM_vector<B,FM_u32>& dimensions,
                               const FM_structured_cell<B>* sc)
{
  assert(sc->get_dimension() == B);
  assert(!sc->is_subsimplex());

  std::vector<FM_ptr<FM_cell> > adjacent_cells;
  for (int i = 0; i < B; i++) {
    FM_u32 hi = dimensions[i] - 2;
    if (sc->get_index(i) < hi) {
      FM_vector<B,FM_u32> indices = sc->get_indices();
      ++indices[i];
      adjacent_cells.push_back(new FM_structured_B_cell<B>
                               (sc->get_time(), sc->get_submesh_id(),
                                indices));
    }
    if (sc->get_index(i) > 0) {
      FM_vector<B,FM_u32> indices = sc->get_indices();
      --indices[i];
      adjacent_cells.push_back(new FM_structured_B_cell<B>
                               (sc->get_time(), sc->get_submesh_id(),
                                indices));
    }
  }
  return adjacent_cells;
}

template <int B>
int FM_structured_mesh_base_to_cell(const FM_vector<B,FM_u32>& dimensions,
                                    const FM_base<B>& b,
                                    FM_ptr<FM_structured_B_cell<B> >* sc)
{
  FM_vector<B,FM_u32> indices;
  for (int i = 0; i < B; i++) {
    if (!(FM_coord(0) <= b[i] && b[i] <= FM_coord(dimensions[i] - 1)))
      return FM_OUT_OF_BOUNDS;
    indices[i] = FM_u32(b[i]);
    if (indices[i] == dimensions[i] - 1) indices[i]--;
  }
  FM_time<FM_u32> time;
  *sc = new FM_structured_B_cell<B>(time, b.submesh_id, indices);
  return FM_OK;
}

template <int B, int D>
int FM_structured_mesh_phys_to_subsimplex
(const FM_phys<D>& p,
 const FM_ptr<FM_structured_B_cell<B> >& sc,
 const FM_vector<D,FM_coord> [],
 FM_context*, FM_ptr<FM_cell>*)
{
  FM_ostringstream err;
  err << "FM_structured_mesh<" << B << "," << D <<
    ">::phys_to_subsimplex(" << p << ", " << *sc << ",,): " <<
    "no simplicial decomposition";
  throw std::logic_error(err.str());
}

template <>
int FM_structured_mesh_phys_to_subsimplex<3,3>
(const FM_phys<3>& p,
 const FM_ptr<FM_structured_B_cell<3> >& sc,
 const FM_vector<3,FM_coord> cv[],
 FM_context* ctxt, FM_ptr<FM_cell>* c)
{
  bool sdo =
    FM_simplicial_decomposition_odd<3>(sc->get_indices(),
                                       ctxt->simplicial_decomposition);
  FM_u32 subid = !sdo ? 0 : 5;
  for (FM_u32 i = 0; i < 4; i++) {
    int orientation =
      FM_orient(cv[FM_subtetrahedron_face[subid][0][0]],
                cv[FM_subtetrahedron_face[subid][0][1]],
                cv[FM_subtetrahedron_face[subid][0][2]], p);
    if (orientation > 0)
      break;
    subid++;
  }
  *c = new FM_structured_subsimplex<3>(sc->get_time(), sc->get_submesh_id(),
                                       3, subid, sc->get_indices());
  return FM_OK;
}


//
// The following are iterator implementations used when
// constructing iterators for structured meshes.  The particular
// implementation choice depends on the type of iteration specified
// by the iter_attrs argument to begin().
//
template <int B>
class FM_structured_mesh_iter_impl : public FM_iter_impl
{
public:
  FM_structured_mesh_iter_impl(const FM_vector<B,FM_u32>& b,
                               const FM_vector<B,FM_u32>& e,
                               const FM_vector<B,FM_u32>& s) :
    begin(b),
    end(e),
    stride(s)
  {}

protected:
  const FM_vector<B,FM_u32> begin, end, stride;
};


template <int B>
class FM_structured_mesh_0_cell_iter_impl:
  public FM_structured_mesh_iter_impl<B>
{
public:
  FM_structured_mesh_0_cell_iter_impl(const FM_time<FM_u32>& time,
                                      const FM_submesh_id& submesh_id,
                                      const FM_vector<B,FM_u32>& b,
                                      const FM_vector<B,FM_u32>& e,
                                      const FM_vector<B,FM_u32>& s) :
    FM_structured_mesh_iter_impl<B>(b, e, s)
  {
    structured_0_cell = new FM_structured_0_cell<B>(time, submesh_id, b);
  }

  FM_structured_mesh_0_cell_iter_impl(const FM_structured_0_cell<B>* sc,
                                      const FM_vector<B,FM_u32>& b,
                                      const FM_vector<B,FM_u32>& e,
                                      const FM_vector<B,FM_u32>& s) :
    FM_structured_mesh_iter_impl<B>(b, e, s), structured_0_cell(sc) {}

  virtual FM_iter_impl* copy() const
  {
    return new FM_structured_mesh_0_cell_iter_impl(structured_0_cell,
                                                   begin, end, stride);
  }

  virtual const FM_cell* advance()
  {
    if (structured_0_cell->get_reference_count() == 1) {
      FM_structured_0_cell<B>* sc = const_cast<FM_structured_0_cell<B>*>
        (static_cast<const FM_structured_0_cell<B>*>(structured_0_cell));
      for (int d = 0; d < B; d++) {
        sc->indices[d] += stride[d];
        if (sc->indices[d] < end[d])
          return structured_0_cell;
        sc->indices[d] = begin[d];
      }
    }
    else {
      FM_vector<B,FM_u32> indices = structured_0_cell->indices;
      for (int d = 0; d < B; d++) {
        indices[d] += stride[d];
        if (indices[d] < end[d]) {
          return new FM_structured_0_cell<B>
            (structured_0_cell->get_time(),
             structured_0_cell->get_submesh_id(),
             indices);
        }
        indices[d] = begin[d];
      }
    }
    return 0;
  }

  virtual const FM_cell* dereference() const
  {
    return structured_0_cell;
  }

protected:
  FM_ptr<FM_structured_0_cell<B> > structured_0_cell;
};


template <int B>
class FM_structured_mesh_B_cell_iter_impl:
  public FM_structured_mesh_iter_impl<B>
{
public:
  FM_structured_mesh_B_cell_iter_impl(const FM_time<FM_u32>& time,
                                      const FM_submesh_id& submesh_id,
                                      const FM_vector<B,FM_u32>& b,
                                      const FM_vector<B,FM_u32>& e,
                                      const FM_vector<B,FM_u32>& s) :
    FM_structured_mesh_iter_impl<B>(b, e, s)
  {
    structured_B_cell = new FM_structured_B_cell<B>(time, submesh_id, b);
  }

  FM_structured_mesh_B_cell_iter_impl(const FM_structured_B_cell<B>* sc,
                                      const FM_vector<B,FM_u32>& b,
                                      const FM_vector<B,FM_u32>& e,
                                      const FM_vector<B,FM_u32>& s) :
    FM_structured_mesh_iter_impl<B>(b, e, s), structured_B_cell(sc) {}

  virtual FM_iter_impl* copy() const
  {
    return new FM_structured_mesh_B_cell_iter_impl(structured_B_cell,
                                                   begin, end, stride);
  }

  virtual const FM_cell* advance()
  {
    if (structured_B_cell->get_reference_count() == 1) {
      FM_structured_B_cell<B>* sc = const_cast<FM_structured_B_cell<B>*>
        (static_cast<const FM_structured_B_cell<B>*>(structured_B_cell));
      for (int d = 0; d < B; d++) {
        sc->indices[d] += stride[d];
        if (sc->indices[d] < end[d])
          return structured_B_cell;
        sc->indices[d] = begin[d];
      }
    }
    else {
      FM_vector<B,FM_u32> indices = structured_B_cell->indices;
      for (int d = 0; d < B; d++) {
```

```
        indices[d] += stride[d];
        if (indices[d] < end[d]) {
          return new FM_structured_B_cell<B>
            (structured_B_cell->get_time(),
             structured_B_cell->get_submesh_id(),
             indices);
        }
        indices[d] = begin[d];
      }
    }
    return 0;
  }

  virtual const FM_cell* dereference() const
  {
    return structured_B_cell;
  }

protected:
  FM_ptr<FM_structured_B_cell<B> > structured_B_cell;
};


template <int B>
class FM_structured_mesh_multi_alignment_iter_impl : public FM_iter_impl
{
public:
  FM_structured_mesh_multi_alignment_iter_impl
  (const FM_time<FM_u32>& time, const FM_submesh_id& submesh_id,
   FM_u32 d,
   const std::vector<FM_vector<B,FM_u32> >& e,
   const FM_vector<B,FM_u32>& s) :
    ends(e),
    stride(s)
  {
    FM_vector<B,FM_u32> indices;
    for (int i = 0; i < B; i++)
      indices[i] = 0;
    structured_k_cell =
      new FM_structured_k_cell<B>(time, submesh_id, d, 0, indices);
  }

  FM_structured_mesh_multi_alignment_iter_impl
  (const FM_structured_k_cell<B>* sc,
   const std::vector<FM_vector<B,FM_u32> >& e,
   const FM_vector<B,FM_u32>& s) :
    structured_k_cell(sc), ends(e), stride(s) {}

  virtual FM_iter_impl* copy() const
  {
    return new
      FM_structured_mesh_multi_alignment_iter_impl<B>(structured_k_cell,
                                                      ends, stride);
  }

  virtual const FM_cell* advance()
  {
    FM_u32 d = structured_k_cell->get_dimension();
    FM_u32 a = structured_k_cell->get_alignment();
    FM_vector<B,FM_u32> indices = structured_k_cell->get_indices();
    int i;
    for (i = 0; i < B; i++) {
      indices[i] += stride[i];
      if (indices[i] < ends[a][i]) {
        structured_k_cell =
          new FM_structured_k_cell<B>(structured_k_cell->get_time(),
                                      structured_k_cell->get_submesh_id(),
                                      d, a, indices);
        return structured_k_cell;
      }
      indices[i] = 0;
    }
    a += 1;
    if (a == ends.size())
      return 0;

    for (i = 0; i < B; i++)
      indices[d] = 0;

    structured_k_cell =
      new FM_structured_k_cell<B>(structured_k_cell->get_time(),
                                  structured_k_cell->get_submesh_id(),
                                  d, a, indices);
    return structured_k_cell;
  }

  virtual const FM_cell* dereference() const
  {
    return structured_k_cell;
  }

protected:
  FM_ptr<FM_structured_k_cell<B> > structured_k_cell;
  const std::vector<FM_vector<B,FM_u32> > ends;
  FM_vector<B,FM_u32> stride;
};


template <int B>
class FM_structured_mesh_subsimplex_iter_impl :
  public FM_structured_mesh_iter_impl<B>
{
public:
  FM_structured_mesh_subsimplex_iter_impl(const FM_time<FM_u32>& time,
                                          const FM_submesh_id& submesh_id,
                                          FM_u32 d,
                                          const FM_vector<B,FM_u32>& b,
                                          const FM_vector<B,FM_u32>& e,
                                          const FM_vector<B,FM_u32>& s,
                                          FM_u32 sd) :
    FM_structured_mesh_iter_impl<B>(b, e, s),
    simplicial_decomposition(sd)
  {
    FM_u32 subid = 0;
    bool sdo = FM_simplicial_decomposition_odd(b, sd);
    if (d == 3)
      subid = sdo ? 5 : 0;
    else
      subid = subid % 4 + (sdo ? 2 : 0);
    structured_subsimplex =
      new FM_structured_subsimplex<B>(time, submesh_id, d, subid, b);
  }

  FM_structured_mesh_subsimplex_iter_impl
  (const FM_structured_subsimplex<B>* ss,
   const FM_vector<B,FM_u32>& b,
   const FM_vector<B,FM_u32>& e,
   const FM_vector<B,FM_u32>& s,
   FM_u32 sd) :
    FM_structured_mesh_iter_impl<B>(b, e, s),
    structured_subsimplex(ss),
    simplicial_decomposition(sd)
  {}

  virtual FM_iter_impl* copy() const
  {
    return new
      FM_structured_mesh_subsimplex_iter_impl(structured_subsimplex,
                                              begin, end, stride,
                                              simplicial_decomposition);
  }

  virtual const FM_cell* advance()
  {
    FM_u32 d = structured_subsimplex->get_dimension();
    FM_u32 subid = structured_subsimplex->get_subid();
    FM_vector<B,FM_u32> indices = structured_subsimplex->get_indices();

    subid += 1;
    if (d == 3) {
      if (!(subid == 5 || subid == 10)) {
        structured_subsimplex =
          new FM_structured_subsimplex<B>
          (structured_subsimplex->get_time(),
           structured_subsimplex->get_submesh_id(),
           d, subid, indices);
        return structured_subsimplex;
      }
    }
    else {
      assert(d == 2);
      if (!(subid % 2 == 0)) {
        structured_subsimplex =
          new FM_structured_subsimplex<B>
          (structured_subsimplex->get_time(),
           structured_subsimplex->get_submesh_id(),
           d, subid, indices);
        return structured_subsimplex;
      }
    }

    int i;
    for (i = 0; i < B; i++) {
      indices[i] += stride[i];
      if (indices[i] < end[i])
        break;
      indices[i] = begin[i];
    }
    if (i == B)
      return 0;

    bool sdo = FM_simplicial_decomposition_odd(indices,
                                               simplicial_decomposition);
    if (d == 3)
      subid = sdo ? 5 : 0;
    else
      subid = subid % 4 + (sdo ? 2 : 0);

    structured_subsimplex =
      new FM_structured_subsimplex<B>
      (structured_subsimplex->get_time(),
       structured_subsimplex->get_submesh_id(),
       d, subid, indices);
    return structured_subsimplex;
  }

  virtual const FM_cell* dereference() const
  {
    return structured_subsimplex;
  }

protected:
  FM_ptr<FM_structured_subsimplex<B> > structured_subsimplex;
  const FM_u32 simplicial_decomposition;
};


template <int B>
FM_iter FM_structured_mesh_begin
(const std::vector<FM_vector<B,FM_u32> > alignment_dimensions[],
   const FM_iter_attrs& ia)
{
  int i;

  // set defaults
  FM_time<FM_u32> time;
  FM_submesh_id submesh_id;
  FM_u32 simplicial_decomposition = 0;
  FM_u32 cell_dimension = 0;
  FM_cell_type_enum cell_type = FM_VERTEX_CELL;
```

```
//bool cell_alignment_given = false;
FM_u32 cell_alignment = 0;
FM_vector<B,FM_u32> begin_indices;
for (i = 0; i < B; i++)
  begin_indices[i] = 0;
FM_vector<B,FM_u32> strides;
for (i = 0; i < B; i++)
  strides[i] = 1;
FM_vector<B,FM_u32> end_indices;

// cell_type and end_indices can be implied by other parameters
FM_vector<B,bool> end_indices_given;
for (i = 0; i < B; i++)
  end_indices_given[i] = false;
bool cell_type_given = false;

int n_ignored = 0;
FM_iter_attrs::const_iterator iter;
for (iter = ia.begin(); iter != ia.end(); ++iter) {
  const FM_iter_attr* it = *iter;
  switch(it->attr) {
  case FM_ITER_ATTR_TIME:
    time = dynamic_cast<const FM_time_iter_attr*>(it)->time;
    break;
  case FM_ITER_ATTR_SUBMESH_ID:
    submesh_id =
      dynamic_cast<const FM_submesh_id_iter_attr*>(it)->submesh_id;
    break;
  case FM_ITER_ATTR_CELL_DIMENSION:
    cell_dimension =
      dynamic_cast<const FM_cell_dimension_iter_attr*>(it)->cell_dimension;
    break;
  case FM_ITER_ATTR_CELL_TYPE:
    cell_type =
      dynamic_cast<const FM_cell_type_iter_attr*>(it)->cell_type;
    cell_type_given = true;
    break;
  case FM_ITER_ATTR_AXIS_BEGIN:
    {
      const FM_axis_begin_iter_attr* axis_begin_iter_attr =
        dynamic_cast<const FM_axis_begin_iter_attr*>(it);
      assert(axis_begin_iter_attr->axis < B);
      begin_indices[axis_begin_iter_attr->axis] =
        axis_begin_iter_attr->index;
    }
    break;
  case FM_ITER_ATTR_AXIS_END:
    {
      const FM_axis_end_iter_attr* axis_end_iter_attr =
        dynamic_cast<const FM_axis_end_iter_attr*>(it);
      assert(axis_end_iter_attr->axis < B);
      end_indices[axis_end_iter_attr->axis] =
        axis_end_iter_attr->index;
    }
    break;
  case FM_ITER_ATTR_AXIS_STRIDE:
    {
      const FM_axis_stride_iter_attr* axis_stride_iter_attr =
        dynamic_cast<const FM_axis_stride_iter_attr*>(it);
      assert(axis_stride_iter_attr->axis < B);
      strides[axis_stride_iter_attr->axis] =
        axis_stride_iter_attr->stride;
    }
    break;
  case FM_ITER_ATTR_SIMPLICIAL_DECOMPOSITION:
    simplicial_decomposition =
      dynamic_cast<const FM_simplicial_decomposition_iter_attr*>(it)->
      simplicial_decomposition;
    break;
  default:
    n_ignored++;
  }
}

// cell_type trumps cell_dimension
if (cell_type_given) {
  cell_dimension = FM_cell_type_to_dimension(cell_type);
  if (cell_type == FM_TRIANGLE_CELL ||
      cell_type == FM_TETRAHEDRON_CELL &&
      simplicial_decomposition == 0)
    simplicial_decomposition = 1;
}

for (i = 0; i < B; i++) {
  if (!end_indices_given[i])
    end_indices[i] =
      alignment_dimensions[cell_dimension][cell_alignment][i];
}

// simplicial decomposition only supported for cells of dimension 2 or 3
if (!(cell_dimension == 2 || cell_dimension == 3))
  simplicial_decomposition = 0;

// construct impl
FM_iter_impl* ii;
if (simplicial_decomposition)
  ii =
    new FM_structured_mesh_subsimplex_iter_impl<B>(time, submesh_id,
                                                   cell_dimension,
                                                   begin_indices,
                                                   end_indices, strides,
                                                   simplicial_decomposition);
else {
  if (cell_dimension == 0) {
    ii = new FM_structured_mesh_0_cell_iter_impl<B>
      (time, submesh_id, begin_indices, end_indices, strides);
  }
  else if (cell_dimension == B) {
    ii = new FM_structured_mesh_B_cell_iter_impl<B>
      (time, submesh_id, begin_indices, end_indices, strides);
    }
    // else if (cell_alignment_given) {
    //}
    else {
      ii = new FM_structured_mesh_multi_alignment_iter_impl<B>
        (time, submesh_id, cell_dimension,
         alignment_dimensions[cell_dimension], strides);
    }
  }

  return FM_iter(ii);
}


template <int B, int D>
FM_ptr<FM_shared_object>
FM_structured_mesh_get_aux(const FM_structured_mesh<B,D>* sm,
                           const std::string& key, FM_u32 pass,
                           const FM_time<FM_u32>* t, const FM_submesh_id* sid)
{
  if (key == "base_dimensions") {
    std::vector<FM_ptr<FM_shared_object> > values(B);
    for (FM_u32 i = 0; i < B; i++)
      values[i] = new FM_simple_value<FM_u32>(sm->dimensions[i]);
    return new FM_tuple_value(values);
  }
  else if (key == "cards") {
    std::vector<FM_ptr<FM_shared_object> > values(B + 1);
    for (FM_u32 i = 0; i <= B; i++)
      values[i] = new FM_simple_value<FM_u64>(sm->get_card(i, 0, t, sid));
    return new FM_tuple_value(values);
  }
  return sm->FM_mesh<B,D>::get_aux(key, pass, t, sid);
}

template <int B, int D>
std::set<std::string>
FM_structured_mesh_get_property_names_aux(const FM_structured_mesh<B,D>* sm,
                                          std::set<std::string>& names,
                                          const FM_time<FM_u32>* t,
                                          const FM_submesh_id* sid)
{
  names.insert("base_dimensions");
  names.insert("cards");
  return sm->FM_mesh<B,D>::get_property_names_aux(names, t, sid);
}


//
// The generic FM_structured_mesh<B,D> declaration:
//
template <int B, int D>
class FM_structured_mesh : public FM_mesh<B,D>
{
public:
  const FM_vector<B,FM_u32> dimensions;

protected:
  void init(const FM_vector<B,FM_u32>& d)
  {
    FM_structured_mesh_init(this, d);
  }

  FM_structured_mesh(FM_properties_cache* pc) :
    FM_mesh<B,D>(pc)
  {}

  FM_structured_mesh(const FM_vector<B,FM_u32>& d,
                     FM_properties_cache* pc) :
    FM_mesh<B,D>(pc),
    dimensions(d)
  {
    init(dimensions);
  }

public:
  virtual FM_u64
  get_card(FM_u32 k, FM_u32 = 0, const FM_time<FM_u32>* = 0,
           const FM_submesh_id* = 0) const
  {
    return FM_structured_mesh_get_card(FM_u32(B), alignment_cards, k);
  }

  virtual FM_vector<B,FM_u32>
  get_base_dimensions(const FM_time<FM_u32>* = 0,
                      const FM_submesh_id* = 0) const
  {
    return dimensions;
  }

  virtual bool
  get_structured_behavior(const FM_time<FM_u32>* = 0,
                          const FM_submesh_id* = 0) const
  {
    return true;
  }

  virtual FM_u64 cell_to_enum(const FM_cell* c) const
  {
    return FM_structured_mesh_cell_to_enum(alignment_dimensions,
                                           alignment_cards, c);
  }

  virtual FM_ptr<FM_cell>
  enum_to_cell(FM_u64 e, FM_u32 d, FM_u32 sd = 0,
               const FM_time<FM_u32>* t = 0,
               const FM_submesh_id* sid = 0) const
  {
    return FM_structured_mesh_enum_to_cell(alignments, alignment_dimensions,
                                           alignment_cards, e, d, sd, t, sid);
```

```
  }

  virtual std::vector<FM_ptr<FM_cell> >
  faces(const FM_cell* c, FM_u32 k) const
  {
    return FM_structured_mesh_faces(dimensions, alignments, c, k);
  }

  virtual std::vector<FM_ptr<FM_cell> >
  adjacencies(const FM_cell* c) const
  {
    const FM_structured_cell<B>* sc =
      dynamic_cast<const FM_structured_cell<B>*>(c);
    if (sc == 0)
      FM_throw_bad_cell_argument(this, "adjacencies", c);
    return FM_structured_mesh_adjacencies(dimensions, sc);
  }

  virtual int
  base_to_cell(const FM_base<B>& b,
               FM_ptr<FM_structured_B_cell<B> >* sc) const
  {
    return FM_structured_mesh_base_to_cell(dimensions, b, sc);
  }

  virtual int
  phys_to_cell(const FM_phys<D>& p, FM_context* ctxt, FM_ptr<FM_cell>* c) const
  {
    FM_base<B> b;
    FM_ptr<FM_structured_B_cell<B> > sc;
    int res = phys_to_base(p, ctxt, &b, &sc);
    *c = sc;
    return res;
  }

  virtual int
  phys_to_subsimplex(const FM_phys<D>& p,
                     const FM_ptr<FM_structured_B_cell<B> >& sc,
                     FM_context* ctxt, FM_ptr<FM_cell>* c) const
  {
    FM_vector<D,FM_coord> cv[1 << B];
    int res = at_cell(sc, cv);
    if (res != FM_OK)
      return res;
    return FM_structured_mesh_phys_to_subsimplex(p, sc, cv, ctxt, c);
  }

  virtual FM_iter begin() const
  {
    FM_iter_attrs iter_attrs;
    return begin(iter_attrs);
  }

  virtual FM_iter begin(const FM_iter_attrs& ia) const
  {
    return FM_structured_mesh_begin(alignment_dimensions, ia);
  }

  virtual FM_ptr<FM_shared_object>
  get_aux(const std::string& key, FM_u32 pass,
          const FM_time<FM_u32>* t, const FM_submesh_id* sid) const
  {
    return FM_structured_mesh_get_aux(this, key, pass, t, sid);
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& names,
                         const FM_time<FM_u32>* t,
                         const FM_submesh_id* sid) const
  {
    return FM_structured_mesh_get_property_names_aux(this, names, t, sid);
  }

  FM_u64 cube_offsets[1 << B];
  std::vector<FM_vector<B,bool> > alignments[B + 1];
  std::vector<FM_vector<B,FM_u32> > alignment_dimensions[B + 1];
  std::vector<FM_u64> alignment_cards[B + 1];
};


//
// The FM_structured_mesh<3,D> partial specialization:  This
// specialization contains the extra code needed to handle
// simplicial decomposition for meshes with base dimensionality 3.
//
template <int D>
class FM_structured_mesh<3,D> : public FM_mesh<3,D>
{
public:
  const FM_vector<3,FM_u32> dimensions;

protected:
  void init(const FM_vector<3,FM_u32>& d)
  {
    FM_structured_mesh_init(this, d);
  }

  FM_structured_mesh(FM_properties_cache* pc) :
    FM_mesh<3,D>(pc)
  {}

  FM_structured_mesh(const FM_vector<3,FM_u32>& d, FM_properties_cache* pc) :
    FM_mesh<3,D>(pc),
    dimensions(d)
  {
    init(dimensions);
  }

public:
  virtual FM_u64
  get_card(FM_u32 k, FM_u32 sd = 0, const FM_time<FM_u32>* t = 0,
           const FM_submesh_id* sid = 0) const
  {
    FM_u64 card = FM_structured_mesh_get_card(3, alignment_cards, k);
    if (sd == 1 || sd == 2) {
      switch(k) {
      case 0:
        break;
      case 1:
        card += get_card(2, 0, t, sid);
        break;
      case 2:
        card *= 2;
        card += 4 * get_card(3, 0, t, sid);
        break;
      case 3:
        card *= 5;
        break;
      default:
        abort();
      }
    }
    return card;
  }

  virtual FM_vector<3,FM_u32>
  get_base_dimensions(const FM_time<FM_u32>* = 0,
                      const FM_submesh_id* = 0) const
  {
    return dimensions;
  }

  virtual bool
  get_structured_behavior(const FM_time<FM_u32>* = 0,
                          const FM_submesh_id* = 0) const
  {
    return true;
  }

  virtual FM_u64 cell_to_enum(const FM_cell* c) const
  {
    return FM_structured_mesh_cell_to_enum(alignment_dimensions,
                                           alignment_cards, c);
  }

  virtual FM_ptr<FM_cell>
  enum_to_cell(FM_u64 e, FM_u32 d, FM_u32 sd = 0,
               const FM_time<FM_u32>* t = 0, const FM_submesh_id* sid = 0) const
  {
    return FM_structured_mesh_enum_to_cell(alignments, alignment_dimensions,
                                           alignment_cards, e, d, sd, t, sid);
  }

  virtual std::vector<FM_ptr<FM_cell> >
  faces(const FM_cell* c, FM_u32 k) const
  {
    return FM_structured_mesh_faces(dimensions, alignments, c, k);
  }

  virtual std::vector<FM_ptr<FM_cell> >
  adjacencies(const FM_cell* c) const
  {
    const FM_structured_cell<3>* sc =
      dynamic_cast<const FM_structured_cell<3>*>(c);
    if (sc == 0)
      FM_throw_bad_cell_argument(this, "adjacencies", c);

    if (!sc->is_subsimplex())
      return FM_structured_mesh_adjacencies(dimensions, sc);

    std::vector<FM_ptr<FM_cell> > adjacent_cells;
    for (int i = 0; i < 4; i++) {
      FM_cell_step cell_step = FM_tetrahedron_step[sc->get_subid()][i];
      FM_vector<3,FM_u32> indices;
      switch (cell_step.dir) {
      case -1:
        if (sc->get_index(cell_step.axis) > 0) {
          indices = sc->get_indices();
          indices[cell_step.axis] -= 1;
          adjacent_cells.push_back
            (new FM_structured_subsimplex<3>(sc->get_time(),
                                             sc->get_submesh_id(), 3,
                                             cell_step.subsimplex,
                                             indices));
        }
        break;
      case 0:
        adjacent_cells.push_back
          (new FM_structured_subsimplex<3>(sc->get_time(),
                                           sc->get_submesh_id(), 3,
                                           cell_step.subsimplex,
                                           sc->get_indices()));
        break;
      case 1:
        if (sc->get_index(cell_step.axis) < dimensions[cell_step.axis] - 2) {
          indices = sc->get_indices();
          indices[cell_step.axis] += 1;
          adjacent_cells.push_back
            (new FM_structured_subsimplex<3>(sc->get_time(),
                                             sc->get_submesh_id(), 3,
                                             cell_step.subsimplex,
                                             indices));
        }
        break;
      default:
        abort();
      }
    }
    return adjacent_cells;
  }
```

```
  virtual int
  base_to_cell(const FM_base<3>& b,
               FM_ptr<FM_structured_B_cell<3> >* sc) const
  {
    return FM_structured_mesh_base_to_cell(dimensions, b, sc);
  }

  virtual int
  phys_to_cell(const FM_phys<D>& p, FM_context* ctxt, FM_ptr<FM_cell>* c) const
  {
    FM_base<3> b;
    FM_ptr<FM_structured_B_cell<3> > sc;
    int res = phys_to_base(p, ctxt, &b, &sc);
    *c = sc;
    return res;
  }

  virtual int
  phys_to_subsimplex(const FM_phys<D>& p,
                     const FM_ptr<FM_structured_B_cell<3> >& sc,
                     FM_context* ctxt, FM_ptr<FM_cell>* c) const
  {
    FM_vector<D,FM_coord> cv[1 << 3];
    int res = at_cell(sc, cv);
    if (res != FM_OK)
      return res;
    return FM_structured_mesh_phys_to_subsimplex(p, sc, cv, ctxt, c);
  }

  virtual FM_iter begin() const
  {
    FM_iter_attrs iter_attrs;
    return begin(iter_attrs);
  }

  virtual FM_iter begin(const FM_iter_attrs& ia) const
  {
    return FM_structured_mesh_begin(alignment_dimensions, ia);
  }

  virtual FM_ptr<FM_shared_object>
  get_aux(const std::string& key, FM_u32 pass,
          const FM_time<FM_u32>* t, const FM_submesh_id* sid) const
  {
    return FM_structured_mesh_get_aux(this, key, pass, t, sid);
  }

  virtual std::set<std::string>
  get_property_names_aux(std::set<std::string>& names,
                         const FM_time<FM_u32>* t,
                         const FM_submesh_id* sid) const
  {
    return FM_structured_mesh_get_property_names_aux(this, names, t, sid);
  }

  FM_u64 cube_offsets[1 << 3];
  std::vector<FM_vector<3,bool> > alignments[3 + 1];
  std::vector<FM_vector<3,FM_u32> > alignment_dimensions[3 + 1];
  std::vector<FM_u64> alignment_cards[3 + 1];
};


//
// The following member function definitions for structured_cell subclasses
// specify the cell's role in computing linear indices in association
// with a structured_mesh.  We provide optimized specializations for the
// base dimensionalities that are most common.
//


template <int B>
void FM_structured_k_cell<B>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  const FM_u32* vi = FM_structured_hexahedron_faces[dimension][alignment];
  *n_ind = FM_pow_2(dimension);
  const FM_structured_mesh<B,B>* sm =
    reinterpret_cast<const FM_structured_mesh<B,B>*>(m);
  FM_u64 index = indices[B - 1];
  for (int i = B - 2; i >= 0; i--) {
    index *= sm->dimensions[i];
    index += indices[i];
  }
  for (FM_u32 i = 0; i < *n_ind; i++)
    ind[i] = index + sm->cube_offsets[vi[i]];
}

template <>
void FM_structured_k_cell<2>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  const FM_u32* vi = FM_structured_hexahedron_faces[dimension][alignment];
  *n_ind = FM_pow_2(dimension);
  const FM_structured_mesh<2,2>* sm =
    reinterpret_cast<const FM_structured_mesh<2,2>*>(m);
  FM_u64 index = indices[1] * sm->dimensions[0] + indices[0];
  for (FM_u32 i = 0; i < *n_ind; i++)
    ind[i] = index + sm->cube_offsets[vi[i]];
}

template <>
void FM_structured_k_cell<3>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  const FM_u32* vi = FM_structured_hexahedron_faces[dimension][alignment];
  *n_ind = FM_pow_2(dimension);
  const FM_structured_mesh<3,3>* sm =
    reinterpret_cast<const FM_structured_mesh<3,3>*>(m);
  FM_u64 index = indices[2] * sm->dimensions[0] * sm->dimensions[1] +
    indices[1] * sm->dimensions[0] + indices[0];
  for (FM_u32 i = 0; i < *n_ind; i++)
    ind[i] = index + sm->cube_offsets[vi[i]];
}

template <int B>
void FM_structured_0_cell<B>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  *n_ind = 1;
  const FM_structured_mesh<B,B>* sm =
    reinterpret_cast<const FM_structured_mesh<B,B>*>(m);
  FM_u64 index = indices[B - 1];
  for (int i = B - 2; i >= 0; i--) {
    index *= sm->dimensions[i];
    index += indices[i];
  }
  ind[0] = index;
}

template <>
void FM_structured_0_cell<1>::
structured_mesh_vertex_indices(const FM_mesh_*, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  *n_ind = 1;
  ind[0] = indices[0];
}

template <>
void FM_structured_0_cell<2>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  *n_ind = 1;
  const FM_structured_mesh<2,2>* sm =
    reinterpret_cast<const FM_structured_mesh<2,2>*>(m);
  ind[0] = indices[1] * sm->dimensions[0] + indices[0];
}

template <>
void FM_structured_0_cell<3>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  *n_ind = 1;
  const FM_structured_mesh<3,3>* sm =
    reinterpret_cast<const FM_structured_mesh<3,3>*>(m);
  ind[0] = indices[2] * sm->dimensions[0] * sm->dimensions[1] +
    indices[1] * sm->dimensions[0] + indices[0];
}

template <int B>
void FM_structured_B_cell<B>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  *n_ind = FM_pow_2(B);
  const FM_structured_mesh<B,B>* sm =
    reinterpret_cast<const FM_structured_mesh<B,B>*>(m);
  FM_u64 index = indices[B - 1];
  for (int i = B - 2; i >= 0; i--) {
    index *= sm->dimensions[i];
    index += indices[i];
  }
  for (FM_u32 i = 0; i < *n_ind; i++)
    ind[i] = index + sm->cube_offsets[i];
}

template <>
void FM_structured_B_cell<3>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  *n_ind = 8;
  const FM_structured_mesh<3,3>* sm =
    reinterpret_cast<const FM_structured_mesh<3,3>*>(m);
  FM_u64 index = indices[2] * sm->dimensions[0] * sm->dimensions[1] +
    indices[1] * sm->dimensions[0] + indices[0];
  for (FM_u32 i = 0; i < 8; i++)
    ind[i] = index + sm->cube_offsets[i];
}

template <int B>
void FM_structured_subsimplex<B>::
structured_mesh_vertex_indices(const FM_mesh_*, FM_u32* n_ind, FM_u64[]) const
{
  *n_ind = 0;
}


template <>
void FM_structured_subsimplex<2>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  const FM_u32* vi = FM_structured_hexahedron_subfaces[dimension][subid];
  *n_ind = dimension + 1;
  const FM_structured_mesh<2,2>* sm =
    reinterpret_cast<const FM_structured_mesh<2,2>*>(m);
  FM_u64 index = indices[1] * sm->dimensions[0] + indices[0];
  for (FM_u32 i = 0; i < *n_ind; i++)
    ind[i] = index + sm->cube_offsets[vi[i]];
}
```

```
template <>
void FM_structured_subsimplex<3>::
structured_mesh_vertex_indices(const FM_mesh_* m, FM_u32* n_ind,
                               FM_u64 ind[]) const
{
  const FM_u32* vi = FM_structured_hexahedron_subfaces[dimension][subid];
  *n_ind = dimension + 1;
  const FM_structured_mesh<3,3>* sm =
    reinterpret_cast<const FM_structured_mesh<3,3>*>(m);
  FM_u64 index = indices[2] * sm->dimensions[0] * sm->dimensions[1] +
    indices[1] * sm->dimensions[0] + indices[0];
  for (FM_u32 i = 0; i < *n_ind; i++)
    ind[i] = index + sm->cube_offsets[vi[i]];
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_SUBMESH_ID_H_
#define _FM_SUBMESH_ID_H_
/*
 * NAME: FM_submesh_id.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <iostream>
#include <stdexcept>
#include "FM_types.h"


class FM_submesh_id
{
public:
  FM_submesh_id() : id(-1) {}
  FM_submesh_id(int i) : id(i) {}
  FM_submesh_id(FM_u32 i) : id(int(i)) {}

  void set(FM_u32 i) { id = int(i); }
  inline bool defined() const { return id != -1; }

  friend std::ostream& operator<<(std::ostream& o, const FM_submesh_id& s)
  {
    if (s.defined())
      o << s.id;
    else
      o << "<submesh_id undefined>";
    return o;
  }

  friend bool operator==(const FM_submesh_id& lhs, const FM_submesh_id& rhs)
  {
    return lhs.id == rhs.id;
  }

  friend bool operator<(const FM_submesh_id& lhs, const FM_submesh_id& rhs)
  {
    return lhs.id < rhs.id;
  }

  FM_u32 index() const
  {
    if (!defined())
      throw std::logic_error("attempting to access undefined submesh_id");
    return FM_u32(id);
  }

private:
  int id;
};

bool operator !=(const FM_submesh_id& lhs, const FM_submesh_id& rhs)
{
  return !(lhs == rhs);
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_TIME_H_
#define _FM_TIME_H_
/*
 * NAME: FM_time.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <iostream>
#include "FM_types.h"


template <typename T>
class FM_time
{
public:
  FM_time() : value(undefined_value) {}
  FM_time(T t) : value(t) {}

  T& operator=(const T& t) { value = t; }
  T get() const
  {
    if (!defined())
      throw std::logic_error("attempting to access undefined time");
    return value;
  }
  void set(T t) { value = t; }
  void set_undefined() { value = undefined_value; }

  inline bool defined() const { return value != undefined_value; }

  friend bool operator==(const FM_time& lhs, const FM_time& rhs)
  {
    return lhs.value == rhs.value;
  }

  friend bool operator!=(const FM_time& lhs, const FM_time& rhs)
  {
    return lhs.value != rhs.value;
  }

  friend bool operator<(const FM_time& lhs, const FM_time& rhs)
  {
    return lhs.value < rhs.value;
  }

  T value;
  static const T undefined_value;
};


template <typename T>
const T FM_time<T>::undefined_value = T(-1e30);

template <>
const FM_u32 FM_time<FM_u32>::undefined_value = 0xFFFFFFFF;


template <typename T>
std::ostream& operator<<(std::ostream& lhs, const FM_time<T>& rhs)
{
  if (rhs.defined())
    return lhs << rhs.get();
  else
    return lhs << "<time undefined>";
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_TIMER_H_
#define _FM_TIMER_H_
/*
 * NAME: FM_timer.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <sys/time.h>

class FM_timer
{
public:
  FM_timer() { reset(); }
  void reset() { total = 0.0; }
  void start() { gettimeofday(&start_tv, (struct timezone *) 0); }
  void stop()
  {
    struct timeval stop_tv;
    gettimeofday(&stop_tv, (struct timezone *) 0);
    long dts = stop_tv.tv_sec - start_tv.tv_sec;
    long dtus = stop_tv.tv_usec - start_tv.tv_usec;
    double dt = (double) dts + (double) dtus * 1.0e-6;

    // round to milliseconds
    long millisec = (long) (dt * 1000.0 + 0.5);
    total += millisec * 1000.0;
  }
  double elapsed() { return total; }

private:
  struct timeval start_tv;
  double total;
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_TYPES_H_
#define _FM_TYPES_H_
/*
 * NAME: FM_types.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#ifndef FM_COORD
#define FM_COORD
typedef float FM_coord;
#endif

typedef unsigned short FM_u16;
typedef unsigned FM_u32;
typedef unsigned long long FM_u64;

struct FM_true_type {};
struct FM_false_type {};

template <int N, typename T> class FM_vector;

template <typename T>
struct FM_traits
{
  typedef T element_type;
  typedef FM_false_type is_scalar;
};

template <>
struct FM_traits<char>
{
  typedef char element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<unsigned char>
{
  typedef unsigned char element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<short>
{
  typedef short element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<unsigned short>
{
  typedef unsigned short element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<int>
{
  typedef int element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<unsigned int>
{
  typedef unsigned int element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<long>
{
  typedef long element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<unsigned long>
{
  typedef unsigned long element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<long long>
{
  typedef long long element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<unsigned long long>
{
  typedef unsigned long long element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<float>
{
  typedef float element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<double>
{
  typedef double element_type;
  typedef FM_true_type is_scalar;
};

template <>
struct FM_traits<long double>
{
  typedef long double element_type;
  typedef FM_true_type is_scalar;
};

template <int N, typename T>
struct FM_traits<FM_vector<N,T> >
{
  typedef typename FM_traits<T>::element_type element_type;
  typedef FM_false_type is_scalar;
};


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_VECTOR_H_
#define _FM_VECTOR_H_
/*
 * NAME: FM_vector.h
 *
 * WRITTEN BY:
 *       Patrick Moran           pmoran@nas.nasa.gov
 */
#include <iostream>
#include <utility>
#if defined(__sgi) && !defined(__GNUC__)
#include <math.h>
#else
#include <cmath>
#endif
#include "FM_types.h"


template <int N, typename T>
T FM_dot(const FM_vector<N,T>&, const FM_vector<N,T>&);

template <typename T>
FM_vector<3,T> FM_cross(const FM_vector<3,T>&, const FM_vector<3,T>&);


template <int N, typename T>
class FM_vector
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    for (int i = 0; i < N; i++)
      d[i] = dat[i];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<N,S>& dat)
  {
    for (int i = 0; i < N; i++)
      d[i] = static_cast<T>(dat[i]);
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  typename FM_traits<T>::element_type* v()
  {
    return reinterpret_cast<typename FM_traits<T>::element_type*>(d);
  }
  const typename FM_traits<T>::element_type* v() const
  {
    return reinterpret_cast<const typename FM_traits<T>::element_type*>(d);
  }

  friend bool operator==(const FM_vector<N,T>& lhs,
                         const FM_vector<N,T>& rhs)
  {
    bool res = true;
    for (int i = 0; i < N; i++) {
      if (!(lhs[i] == rhs[i])) {
        res = false;
        break;
      }
    }
    return res;
  }

  FM_vector<N,T>& operator+=(const FM_vector<N,T>& v) {
    for (int i = 0; i < N; i++)
      d[i] += v[i];
    return *this;
  }

  FM_vector<N,T>& operator-=(const FM_vector<N,T>& v) {
    for (int i = 0; i < N; i++)
      d[i] -= v[i];
    return *this;
  }

  FM_vector<N,T>& operator*=(typename FM_traits<T>::element_type s) {
    for (int i = 0; i < N; i++)
      d[i] *= s;
    return *this;
  }

  FM_vector<N,T>& operator/=(typename FM_traits<T>::element_type s) {
    for (int i = 0; i < N; i++)
      d[i] /= s;
    return *this;
  }

  friend FM_vector<N,T> operator-(const FM_vector<N,T>& u)
  {
    T tmp[N];
    for (int i = 0; i < N; i++)
      tmp[i] = -u[i];
    return FM_vector<N,T>(tmp);
  }

  friend FM_vector<N,T> operator+(const FM_vector<N,T>& lhs,
                                  const FM_vector<N,T>& rhs)
  {
    T tmp[N];
    for (int i = 0; i < N; i++)
      tmp[i] = lhs[i] + rhs[i];
    return FM_vector<N,T>(tmp);
  }
  friend FM_vector<N,T> operator-(const FM_vector<N,T>& lhs,
                                  const FM_vector<N,T>& rhs)
  {
    T tmp[N];
    for (int i = 0; i < N; i++)
      tmp[i] = lhs[i] - rhs[i];
    return FM_vector<N,T>(tmp);
  }

  friend FM_vector<N,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<N,T>& rhs)
  {
    T tmp[N];
    for (int i = 0; i < N; i++)
      tmp[i] = lhs * rhs[i];
    return FM_vector<N,T>(tmp);
  }

  friend FM_vector<N,T>
  operator*(const FM_vector<N,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    T tmp[N];
    for (int i = 0; i < N; i++)
      tmp[i] = lhs[i] * rhs;
    return FM_vector<N,T>(tmp);
  }

private:
  T d[N];
};


template <typename T>
class FM_vector<1,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<1,S>& dat)
  {
    d[0] = static_cast<T>(dat[0]);
  }
  FM_vector(const T& a0)
  {
    d[0] = a0;
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  typename FM_traits<T>::element_type* v()
  {
    return reinterpret_cast<typename FM_traits<T>::element_type*>(d);
  }
  const typename FM_traits<T>::element_type* v() const
  {
    return reinterpret_cast<const typename FM_traits<T>::element_type*>(d);
  }

  friend bool operator==(const FM_vector<1,T>& lhs,
                         const FM_vector<1,T>& rhs)
  {
    return
      lhs.d[0] == rhs.d[0];
  }

  FM_vector<1,T>& operator+=(const FM_vector<1,T>& v)
  {
    d[0] += v[0];
    return *this;
  }

  FM_vector<1,T>& operator-=(const FM_vector<1,T>& v)
  {
    d[0] -= v[0];
    return *this;
  }

  FM_vector<1,T>& operator*=(typename FM_traits<T>::element_type s)
  {
    d[0] *= s;
    return *this;
  }

  FM_vector<1,T>& operator/=(typename FM_traits<T>::element_type s)
  {
    d[0] /= s;
    return *this;
  }

  friend FM_vector<1,T> operator-(const FM_vector<1,T>& u)
  {
    return FM_vector<1,T>(-u.d[0]);
  }

  friend FM_vector<1,T> operator+(const FM_vector<1,T>& lhs,
                                  const FM_vector<1,T>& rhs)
  {
    return FM_vector<1,T>(lhs.d[0] + rhs.d[0]);
  }
  friend FM_vector<1,T> operator-(const FM_vector<1,T>& lhs,
                                  const FM_vector<1,T>& rhs)
  {
    return FM_vector<1,T>(lhs.d[0] - rhs.d[0]);
  }
```

```
  friend FM_vector<1,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<1,T>& rhs)
  {
    return FM_vector<1,T>(lhs * rhs.d[0]);
  }
  friend FM_vector<1,T>
  operator*(const FM_vector<1,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    return FM_vector<1,T>(lhs.d[0] * rhs);
  }

  friend T FM_dot<T>(const FM_vector<1,T>&,
                     const FM_vector<1,T>&);

private:
  T d[1];
};


template <typename T>
class FM_vector<2,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
    d[1] = dat[1];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<2,S>& dat)
  {
    d[0] = static_cast<T>(dat[0]);
    d[1] = static_cast<T>(dat[1]);
  }
  FM_vector(const T& a0, const T& a1)
  {
    d[0] = a0;
    d[1] = a1;
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  typename FM_traits<T>::element_type* v()
  {
    return reinterpret_cast<typename FM_traits<T>::element_type*>(d);
  }
  const typename FM_traits<T>::element_type* v() const
  {
    return reinterpret_cast<const typename FM_traits<T>::element_type*>(d);
  }

  friend bool operator==(const FM_vector<2,T>& lhs,
                         const FM_vector<2,T>& rhs)
  {
    return
      lhs.d[0] == rhs.d[0] &&
      lhs.d[1] == rhs.d[1];
  }

  FM_vector<2,T>& operator+=(const FM_vector<2,T>& v)
  {
    d[0] += v[0];
    d[1] += v[1];
    return *this;
  }

  FM_vector<2,T>& operator-=(const FM_vector<2,T>& v)
  {
    d[0] -= v[0];
    d[1] -= v[1];
    return *this;
  }

  FM_vector<2,T>& operator*=(typename FM_traits<T>::element_type s)
  {
    d[0] *= s;
    d[1] *= s;
    return *this;
  }

  FM_vector<2,T>& operator/=(typename FM_traits<T>::element_type s)
  {
    d[0] /= s;
    d[1] /= s;
    return *this;
  }

  friend FM_vector<2,T> operator-(const FM_vector<2,T>& u)
  {
    return FM_vector<2,T>(-u.d[0], -u.d[1]);
  }

  friend FM_vector<2,T> operator+(const FM_vector<2,T>& lhs,
                                  const FM_vector<2,T>& rhs)
  {
    return FM_vector<2,T>(lhs.d[0] + rhs.d[0],
                          lhs.d[1] + rhs.d[1]);
  }
  friend FM_vector<2,T> operator-(const FM_vector<2,T>& lhs,
                                  const FM_vector<2,T>& rhs)
  {
    return FM_vector<2,T>(lhs.d[0] - rhs.d[0],
                          lhs.d[1] - rhs.d[1]);
  }

  friend FM_vector<2,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<2,T>& rhs)
  {
    return FM_vector<2,T>(lhs * rhs.d[0],
                          lhs * rhs.d[1]);
  }
  friend FM_vector<2,T>
  operator*(const FM_vector<2,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    return FM_vector<2,T>(lhs.d[0] * rhs,
                          lhs.d[1] * rhs);
  }

  friend T FM_dot<T>(const FM_vector<2,T>&,
                     const FM_vector<2,T>&);

private:
  T d[2];
};


template <typename T>
class FM_vector<3,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
    d[1] = dat[1];
    d[2] = dat[2];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<3,S>& dat)
  {
    d[0] = static_cast<T>(dat[0]);
    d[1] = static_cast<T>(dat[1]);
    d[2] = static_cast<T>(dat[2]);
  }
  explicit FM_vector(const FM_vector<4,T>& v)
  {
    d[0] = v[0];
    d[1] = v[1];
    d[2] = v[2];
  }
  FM_vector(const T& a0, const T& a1, const T& a2)
  {
    d[0] = a0;
    d[1] = a1;
    d[2] = a2;
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  typename FM_traits<T>::element_type* v()
  {
    return reinterpret_cast<typename FM_traits<T>::element_type*>(d);
  }
  const typename FM_traits<T>::element_type* v() const
  {
    return reinterpret_cast<const typename FM_traits<T>::element_type*>(d);
  }

  friend bool operator==(const FM_vector<3,T>& lhs,
                         const FM_vector<3,T>& rhs)
  {
    return
      lhs.d[0] == rhs.d[0] &&
      lhs.d[1] == rhs.d[1] &&
      lhs.d[2] == rhs.d[2];
  }

  FM_vector<3,T>& operator+=(const FM_vector<3,T>& v)
  {
    d[0] += v[0];
    d[1] += v[1];
    d[2] += v[2];
    return *this;
  }

  FM_vector<3,T>& operator-=(const FM_vector<3,T>& v)
  {
    d[0] -= v[0];
    d[1] -= v[1];
    d[2] -= v[2];
    return *this;
  }

  FM_vector<3,T>& operator*=(typename FM_traits<T>::element_type s)
  {
    d[0] *= s;
    d[1] *= s;
    d[2] *= s;
    return *this;
  }

  FM_vector<3,T>& operator/=(typename FM_traits<T>::element_type s)
  {
    d[0] /= s;
    d[1] /= s;
    d[2] /= s;
    return *this;
  }

  friend FM_vector<3,T> operator-(const FM_vector<3,T>& u)
  {
    return FM_vector<3,T>(-u.d[0], -u.d[1], -u.d[2]);
  }
```

```
  friend FM_vector<3,T> operator+(const FM_vector<3,T>& lhs,
                                  const FM_vector<3,T>& rhs)
  {
    return FM_vector<3,T>(lhs.d[0] + rhs.d[0],
                          lhs.d[1] + rhs.d[1],
                          lhs.d[2] + rhs.d[2]);
  }
  friend FM_vector<3,T> operator-(const FM_vector<3,T>& lhs,
                                  const FM_vector<3,T>& rhs)
  {
    return FM_vector<3,T>(lhs.d[0] - rhs.d[0],
                          lhs.d[1] - rhs.d[1],
                          lhs.d[2] - rhs.d[2]);
  }

  friend FM_vector<3,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<3,T>& rhs)
  {
    return FM_vector<3,T>(lhs * rhs.d[0],
                          lhs * rhs.d[1],
                          lhs * rhs.d[2]);
  }
  friend FM_vector<3,T>
  operator*(const FM_vector<3,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    return FM_vector<3,T>(lhs.d[0] * rhs,
                          lhs.d[1] * rhs,
                          lhs.d[2] * rhs);
  }

  friend T FM_dot<T>(const FM_vector<3,T>&,
                     const FM_vector<3,T>&);
  friend FM_vector<3,T> FM_cross<T>(const FM_vector<3,T>&,
                                    const FM_vector<3,T>&);

private:
  T d[3];
};


template <typename T>
class FM_vector<4,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
    d[1] = dat[1];
    d[2] = dat[2];
    d[3] = dat[3];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<4,S>& dat)
  {
    d[0] = static_cast<T>(dat[0]);
    d[1] = static_cast<T>(dat[1]);
    d[2] = static_cast<T>(dat[2]);
    d[3] = static_cast<T>(dat[3]);
  }
  FM_vector(const T& a0, const T& a1, const T& a2, const T& a3)
  {
    d[0] = a0;
    d[1] = a1;
    d[2] = a2;
    d[3] = a3;
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  typename FM_traits<T>::element_type* v()
  {
    return reinterpret_cast<typename FM_traits<T>::element_type*>(d);
  }
  const typename FM_traits<T>::element_type* v() const
  {
    return reinterpret_cast<const typename FM_traits<T>::element_type*>(d);
  }

  friend bool operator==(const FM_vector<4,T>& lhs,
                         const FM_vector<4,T>& rhs)
  {
    return
      lhs.d[0] == rhs.d[0] &&
      lhs.d[1] == rhs.d[1] &&
      lhs.d[2] == rhs.d[2] &&
      lhs.d[3] == rhs.d[3];
  }

  FM_vector<4,T>& operator+=(const FM_vector<4,T>& v)
  {
    d[0] += v[0];
    d[1] += v[1];
    d[2] += v[2];
    d[3] += v[3];
    return *this;
  }

  FM_vector<4,T>& operator-=(const FM_vector<4,T>& v)
  {
    d[0] -= v[0];
    d[1] -= v[1];
    d[2] -= v[2];
    d[3] -= v[3];
    return *this;
  }

  FM_vector<4,T>& operator*=(typename FM_traits<T>::element_type s)
  {
    d[0] *= s;
    d[1] *= s;
    d[2] *= s;
    d[3] *= s;
    return *this;
  }

  FM_vector<4,T>& operator/=(typename FM_traits<T>::element_type s)
  {
    d[0] /= s;
    d[1] /= s;
    d[2] /= s;
    d[3] /= s;
    return *this;
  }

  friend FM_vector<4,T> operator-(const FM_vector<4,T>& u)
  {
    return FM_vector<4,T>(-u.d[0], -u.d[1], -u.d[2], -u.d[3]);
  }

  friend FM_vector<4,T> operator+(const FM_vector<4,T>& lhs,
                                  const FM_vector<4,T>& rhs)
  {
    return FM_vector<4,T>(lhs.d[0] + rhs.d[0],
                          lhs.d[1] + rhs.d[1],
                          lhs.d[2] + rhs.d[2],
                          lhs.d[3] + rhs.d[3]);
  }
  friend FM_vector<4,T> operator-(const FM_vector<4,T>& lhs,
                                  const FM_vector<4,T>& rhs)
  {
    return FM_vector<4,T>(lhs.d[0] - rhs.d[0],
                          lhs.d[1] - rhs.d[1],
                          lhs.d[2] - rhs.d[2],
                          lhs.d[3] - rhs.d[3]);
  }

  friend FM_vector<4,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<4,T>& rhs)
  {
    return FM_vector<4,T>(lhs * rhs.d[0],
                          lhs * rhs.d[1],
                          lhs * rhs.d[2],
                          lhs * rhs.d[3]);
  }
  friend FM_vector<4,T>
  operator*(const FM_vector<4,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    return FM_vector<4,T>(lhs.d[0] * rhs,
                          lhs.d[1] * rhs,
                          lhs.d[2] * rhs,
                          lhs.d[3] * rhs);
  }

  friend T FM_dot<T>(const FM_vector<4,T>&,
                     const FM_vector<4,T>&);

private:
  T d[4];
};


template <int N, typename T>
bool operator!=(const FM_vector<N,T>& lhs,
                const FM_vector<N,T>& rhs)
{
  return !(lhs == rhs);
}


template <int N, typename T>
std::ostream& operator<<(std::ostream& lhs, const FM_vector<N,T>& rhs)
{
  lhs << "(";
  int i;
  for (i = 0; i < N; i++) {
    if (i > 0) lhs << ", ";
    lhs << rhs[i];
  }
  return lhs << ")";
}


template <int N, typename T>
T FM_dot(const FM_vector<N,T>& lhs, const FM_vector<N,T>& rhs)
{
  T res = lhs[0] * rhs[0];
  for (int i = 1; i < N; i++)
    res += lhs[i] * rhs[i];
  return res;
}


template <typename T>
T FM_dot(const FM_vector<1,T>& lhs,
         const FM_vector<1,T>& rhs)
{
  return
    lhs.d[0] * rhs.d[0];
}

template <typename T>
T FM_dot(const FM_vector<2,T>& lhs,
         const FM_vector<2,T>& rhs)
```

```
{
  return
    lhs.d[0] * rhs.d[0] +
    lhs.d[1] * rhs.d[1];
}

template <typename T>
T FM_dot(const FM_vector<3,T>& lhs,
         const FM_vector<3,T>& rhs)
{
  return
    lhs.d[0] * rhs.d[0] +
    lhs.d[1] * rhs.d[1] +
    lhs.d[2] * rhs.d[2];
}

template <typename T>
T FM_dot(const FM_vector<4,T>& lhs,
         const FM_vector<4,T>& rhs)
{
  return
    lhs.d[0] * rhs.d[0] +
    lhs.d[1] * rhs.d[1] +
    lhs.d[2] * rhs.d[2] +
    lhs.d[3] * rhs.d[3];
}


template <typename T>
FM_vector<3,T> FM_cross(const FM_vector<3,T>& lhs,
                        const FM_vector<3,T>& rhs)
{
  return FM_vector<3,T>(lhs.d[1] * rhs.d[2] - rhs.d[1] * lhs.d[2],
                        rhs.d[0] * lhs.d[2] - lhs.d[0] * rhs.d[2],
                        lhs.d[0] * rhs.d[1] - rhs.d[0] * lhs.d[1]);
}


template <int N, typename T>
T FM_mag(const FM_vector<N,T>& v)
{
  return (T) sqrt(FM_dot(v, v));
}


template <int N, typename T>
T FM_distance2(const FM_vector<N,T>& lhs, const FM_vector<N,T>& rhs)
{
  FM_vector<N,T> d = rhs - lhs;
  return FM_dot(d, d);
}


template <int N, typename T>
inline std::pair<FM_vector<N,T>,FM_vector<N,T> >&
FM_operator_min_max_equals(std::pair<FM_vector<N,T>,FM_vector<N,T> >& mm,
                           const FM_vector<N,T>& v)
{
  for (int i = 0; i < N; i++) {
    if (v[i] < mm.first[i])
      mm.first[i] = v[i];
    else if (mm.second[i] < v[i])
      mm.second[i] = v[i];
  }
  return mm;
}


template <int N>
FM_vector<N,bool> operator!(const FM_vector<N,bool>& u)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = !u[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
FM_vector<N,bool> operator&&(const FM_vector<N,bool>& lhs,
                             const FM_vector<N,bool>& rhs)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] && rhs[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
FM_vector<N,bool> operator||(const FM_vector<N,bool>& lhs,
                             const FM_vector<N,bool>& rhs)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] || rhs[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
FM_vector<N,bool> operator^(const FM_vector<N,bool>& lhs,
                            const FM_vector<N,bool>& rhs)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] ^ rhs[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
bool operator<=(const FM_vector<N,bool>& lhs,
                const FM_vector<N,bool>& rhs)
{
  bool res = true;
  for (int i = 0; i < N; i++) {
    if (lhs[i] && !rhs[i]) {
      res = false;
      break;
    }
  }
  return res;
}

template <int N>
bool operator>=(const FM_vector<N,bool>& lhs,
                const FM_vector<N,bool>& rhs)
{
  bool res = true;
  for (int i = 0; i < N; i++) {
    if (!lhs[i] && rhs[i]) {
      res = false;
      break;
    }
  }
  return res;
}


typedef FM_vector<2,int> FM_vector2i;
typedef FM_vector<2,float> FM_vector2f;
typedef FM_vector<2,double> FM_vector2d;
typedef FM_vector<3,int> FM_vector3i;
typedef FM_vector<3,float> FM_vector3f;
typedef FM_vector<3,double> FM_vector3d;
typedef FM_vector<4,int> FM_vector4i;
typedef FM_vector<4,float> FM_vector4f;
typedef FM_vector<4,double> FM_vector4d;


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
```