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 \( \mathbb{R}^D \) (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 \( \mathbb{R}^2 \) and discuss how the same design abstractions would work equally well with other adaptive-grid variations, including meshes in \( \mathbb{R}^3 \).

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 \( \mathbb{R}^D \), also known as physical space. Objects in \( \mathbb{R}^D \) are also said to have a physical dimensionality of \( D \). The regions in \( \mathbb{R}^D \) where fields are defined are discretized by meshes, which in turn are composed of cells. A \( k \)-cell is a subset of \( \mathbb{R}^D \) 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 \( \sigma \) has a set of vertices. We use a more general concept of face than some are familiar with: a face of \( \sigma \) is specified by a non-empty subset of the vertices of \( \sigma \). 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

1If a cell \( \sigma \) is not a simplex, then not every subset of the vertices of \( \sigma \) constitutes a face. In practice it is clear which subsets define valid faces.
with general dimension. A mesh $\mathcal{M}$ is a finite set of cells such that if $\sigma \in \mathcal{M}$, and $\tau$ is a face of $\sigma$, 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 $\text{FM}$. Note that $\text{FM}$ 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 $\text{FM}$, 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 $\text{FM}$ 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[3, 1]). Beginning with Haber et al [8], the fiber bundle model was adapted into a model that supported 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 \( \mathbb{R}^3 \) 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.

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 Wind-tunnel [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.fieldInterface<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,FH.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 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 function 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 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_structuredmesh<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 ≠ 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 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_multifield<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

---

1Note that graph nodes and field nodes are different concepts.
at_cell calls on the underlying fields.

The FM_field<BM, 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 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<BM, D> and FM.time_series.field<BM, 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.
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 Mütte for the interlinked tori plot 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


<table>
<thead>
<tr>
<th>Mesh</th>
<th>Bounding Box</th>
<th>Edge Drawing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abstract</td>
<td>Direct</td>
</tr>
<tr>
<td>Initial</td>
<td>23.5</td>
<td>4.3</td>
</tr>
<tr>
<td>Level 1</td>
<td>343.8</td>
<td>264.4</td>
</tr>
<tr>
<td>Level 2</td>
<td>462.8</td>
<td>35.7</td>
</tr>
<tr>
<td>Level 3</td>
<td>152.4</td>
<td>11.7</td>
</tr>
<tr>
<td>Level 4</td>
<td>90512.6</td>
<td>58007.2</td>
</tr>
</tbody>
</table>

Table 2: Initial FM example timings, in msec. The “Bounding Box” columns illustrate the 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%.


Appendix A

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

/*
• NAME: FMbase.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<T>& t0, const FM submesh id& sid0) :
 FM_vector<B, T>(v), time(t0), submesh id(sid0) {
};

};

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, const FM time<T>& 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, 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, const FM time<T>& 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 << "(\n";
 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, excluding
 * 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
class FM cell : public FM shared object

protected:
FM structured cell();
FM structured cell(const FM vector<B, FM u32> k) : indicies(k) {};
FM structured cell(const FM time<FM u32> t, const FM submesh id s, const FM vector<B, FM u32> k) :
FM cell(t, s, k), indices(k) {};
friend class FM structured mesh<0> cell iter impl<B>;
friend class FM structured mesh<3> cell iter impl<B>;

public:
virtual FM time<FM u32> get time() const { return time; }
virtual FM submesh id get submesh id() const { return submesh id; }
virtual void get submesh id(const FM submesh id& sid) const { return submesh id = sid; }
virtual void set time(const FM time<FM u32>& t) { time = t; }
virtual void set submesh id(const FM submesh id& sid) { submesh id = sid; }
virtual bool is subsimplex() const { return is subsimplex(); }
virtual FM u32 get n faces(FM u32 k) const = 0;
virtual FM u32 get dimension() const = 0;
virtual FM cell type enum get type() const = 0;
virtual void set subid(FM u32)
protected:
FM submesh id get submesh id() const { return submesh id; }

};
```cpp
public:
{
    class FM structured 0 cell : public FM structured cell<B>
    {
        template <int B>
        {
            protected:
                FM_vector<B, FM u32> indices;
            }
        }
    }

    template <int B>
    class FM structured k cell : public FM structured cell<B>
    {
        template <int B>
        {
            protected:
                FM structured k cell() : dimension(k), subid(s) {
                    if (skc) {
                        const FM structured k cell<B>* skc =
                            dynamic_cast<const FM structured k cell<B>*>(&c);
                        if (dimension == skc->dimension &&
                            submesh_id == skc->submesh_id &&
                            time == skc->time) {
                            return true;
                        } else if (dimension == B) {
                            return false;
                        }
                    } else if (sBc) {
                        const FM structured B cell<B>* sBc =
                            dynamic_cast<const FM structured B cell<B>*>(&c);
                        if (skc) {
                            const FM structured 0 cell<B>* s0c =
                                dynamic_cast<const FM structured 0 cell<B>*>(&c);
                            if (alignment) {
                                return skc->alignment == s0c->alignment;
                            } else if (dimension == B) {
                                return s0c->get dimension() == skc->get dimension();
                            } else {
                                return false;
                            }
                        } else if (dimension == B) {
                            return sBc->get dimension() == skc->get dimension();
                        }
                    } else {
                        return false;
                    }
                }

                virtual void set alignment(FM u32 a) {
                    dimension = a;
                    if (skc) {
                        const FM structured k cell<B>* skc =
                            dynamic_cast<const FM structured k cell<B>*>(&c);
                        if (dimension == skc->dimension) {
                            return true;
                        } else if (dimension == B) {
                            return false;
                        }
                    }
                }

                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;
            }
        }
    }

    class FM structured B cell : public FM structured cell<B>
    {
        template <int B>
        {
            protected:
                FM structured B cell() {
                    return false;
                }

                virtual bool isequalto(const FM cell& c) const
                    return dynamic_cast<const FM structured B cell<B>*>(&c);
            }
        }
    }

    template <int B>
    class FM structured _cell : public FM structured cell<B>
    {
        public:
            FM structured _cell(FM u32 d, FM u32 a, const FM vector<B, FM u32>& i)
            : dimension(d), subid(a), indices(i) {
            }

            virtual bool is_equalto(const FM cell& c) const
                return dynamic_cast<const FM structured _cell<B>*>(&c);
            virtual FM u32 get dimension() const {
                return dimension;
            }
        }
    }

    template <int B>
    class FM structured subsimplex : public FM structured cell<B>
    {
        public:
            FM structured subsimplex(const FM mesh*, FM u32*, FM u64[]) const;
            private:
                FM u32 alignment;
                }
        }
    }

    template <int B>
    class FM structured B cell : public FM structured cell<B>
    {
        template <int B>
        {
            protected:
                FM structured B cell(const FM time<FM u32>& t,
                    const FM submesh id& sid, const FM vector<B, FM u32>& i)
            : dimension(d), subid(s), indices(i) {
            }

            virtual bool is_equal_to(const FM cell& c) const
                return dynamic_cast<const FM structured B cell<B>*>(&c);
            virtual FM u32 get dimension() const {
                return dimension;
            }
        }
    }

    class FM structured _cell : public FM structured cell<B>
    {
        protected:
            FM structured _cell() {
                return false;
            }

            virtual bool is_equalto(const FM cell& c) const
                return dynamic_cast<const FM structured _cell<B>*>(&c);
            virtual FM u32 get dimension() const {
                return dimension;
            }
        }
    }
```
return
time = ss-time 44
submesh_id = ss-submesh_id 44
dimension = ss-dimension 44
pubid = ss-pubid 44
indices = ss-indices;
}
return false;
|
virtual void structured_msh_vertex_indices(const FM u32* FM u32, FM u64[]) const;
private:
FM u32 dimension;
FM u32 pubid;
};
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, d, dimension), index(i) |
FM unstructured_cell(const FM_unstructured_cells& uc, FM u32 d, FM u64 i) :
FM cell(uc), dimension(d), index(i) |
FM u32 get_dimension() const { return dimension; }
};
class FM unstructured_pyramid : public FM unstructured_cell
public:
FM unstructured_pyramid(const FM unstructured_cell& uc, FM u64 i):
FM unstructured_pyramid(const FM time<FM u32>& t, const FM submesh_id& s,
FM u32 d, FM u64 i) :
FM u64 i) :
FM u32 dimension;
FM u32 pubid;
};
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 cell(t, d, dimension), index(i) |
FM u32 get_dimension() const { return dimension; }
};
class FH unstructured_cell : public FH cell
public:
virtual std::ostream& str(std::ostream& o) const
virtual FM u32 get n_faces(FM u32 k) const
virtual FM cell type enum get type() const { return FM PYRAMID CELL; }
FM unstructured_pyramid(const FM unstructured_cell& uc, FM u64 i):
FM u64 i) :
FM u32 dimension;
FM u32 pubid;
};
class FM unstructured_prism : public FM unstructured_cell
public:
FM unstructured_prism(const FM time<FM u32>& t, const FM submesh_ids a,
FM u64 i) :
FM unstructured_cell(t, i, 3, 4) |
FM unstructured_prism(const FM_unstructured_cells& uc, FM u32 d, FM u64 i) :
FM unstructured_cell(uc, d, i) |
virtual FM cell_type_enum get_type() const { return FM PRISMA CELL; }
FM u32 get n_faces(FM u32 k) const
FM u32 dimension;
FM u32 pubid;
};
class FM unstructured_hexahedron : public FM unstructured_cell
public:
FM unstructured_hexahedron(const FM time<FM u32>& t, const FM submesh_ids a,
FM u64 i) :
FM u64 i) :
FM u32 dimension;
FM u32 pubid;
};
class FM unstructured_hexahedron : public FM unstructured_cell
public:
FM unstructured_hexahedron(const FM time<FM u32>& t, const FM submesh_ids a,
FM u64 i) :
FM u64 i) :
FM u32 dimension;
FM u32 pubid;
};
* 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.

*/

#ifndef FMCOMBINATORICS_H
#define FMCOMBINATORICS_H

#include <vector>
#include "FMvector.h"

// Emacs mode * c++ *

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

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

template <typename T>
T FMchoose (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}
    };
    return (b < LUT_SIZE) ? lut[b][k] : FMfact(b) / (FMfact(b-k) * FMfact(k));
}

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

// Use code -*- c++ -*-
#include "FM_COMBINATORICS_I_
#include "FM_COMBINATORICS_H"
*/

/* NAME: FM combinatorics.h
 * WRITTEN BY: Patrick Moran pmoran@nas.nasa.gov
 */

#include <vector>
#include "FMvector.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}
    };
    return (b < LUT_SIZE) ? lut[b][k] : FM_fact(b) / (FM_fact(b-k) * FM_fact(k));
}

template <typename T, int N, int B, int K>
void FM_choose(const FM u32 k, std::vector<FMvector<B,bool> >* choices)
{
    int i, ik = int(k);
    if (ik <= B)
        FM u32 nchoices = FM_choose(FM u32(b), k);
    // assert(k < b);
    return b < N? lut[b][k] : FM_fact(b) / (FM_fact(b-k) * FM_fact(k));
}

template <typename T, int N, int B, int K>
void FM_choose(const FM u32 k, std::vector<FMvector<B,bool> >* choices)
{
    int i, ik = int(k);
    if (ik <= B)
        FM u32 nchoices = FM_choose(FM u32(b), k);
    // assert(k < b);
    return b < N? lut[b][k] : FM_fact(b) / (FM_fact(b-k) * FM_fact(k));
}

template <typename T, int N, int B, int K>
void FM_choose(const FM u32 k, std::vector<FMvector<B,bool> >* choices)
{
    int i, ik = int(k);
    if (ik <= B)
        FM u32 nchoices = FM_choose(FM u32(b), k);
    // assert(k < b);
    return b < N? lut[b][k] : FM_fact(b) / (FM_fact(b-k) * FM_fact(k));
}

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

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

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

template <int N, typename T>
bool FM_addition(const FMvector<T>& a, const FMvector<T>& b)
{
    T sum = 0;
    for (int i = 0; i < N; i++)
        sum += a[i];
    return sum & i ? true : false;
}
// Do not mode "---" // */

#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 m, typename T>
class FM constant field : public FM field<B, m, T>
{
public:
    const T constant;
    FM constant field(const FM ptr<FM mesh<B, m>>&, const T& c, int na_, FM properties cache* pc) :
    FM field<B, m, T>(m, na_, pc) :
    constant(c) {
    }

    virtual std::ostream& str(std::ostream& o) const
        return o << "FM constant field<" << B << ", " << m << ", " << typeid(T).name() << ">";

    virtual int at cell(const FM cell*, 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[i] = constant;
        return FM_OK;
    }

    virtual int at cell(const FM cell*, 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;
    }

};

#endif
```cpp
// Emacs mode * c++ *
#ifndef FMCONTEXT_H
#define FMCONTEXT_H
/*
 * NAME: FM context.h
 * WRITTEN BY:
 * Patrick Moran pmoran@nas.nasa.gov
 */
#include "FMcell.h"

class FMcontext
{
public:
    FM context() :
        simplicial_decomposition(0),
        locate_verbosity(0),
        locate_effort(4)
    {
        FM u32 simplicial_decomposition;
        FMptr<FMcell> 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
```

```cpp
// 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,m,T>
{
public:
    FM core field(const FM ptr<FM mesh<B,m> >& mesh,
                   FM u32 na, FM properties cache* pc) :
        FM field<B,m,T_(mesh, na, pc) {
            virtual std::ostream& str(std::ostream& o) const
            return o << "FM core field<" << B << "," << D << "," <<
                     typeid(T).nameO <3 ">";
        }

template <int B, int D, typename T>
class FM core fieldT layout;

template <int B, int D, typename T>
std::pair<T,T>
FM get min max aux (const FMptr<FM core fieldT layout<B, D,T> >& field,
                    const FMtime<FMu32>* t, const FM submesh id* sid,
                    const FMtruetype& 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 + i;
    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>
bool blank checking(const FM ptr<FM core fieldT layout<B, D,T> >& field,
                    const FM context* c, const FM submesh id* sid,
                    const FM_true types tti)
{
    field->get simple value("Blank checking", &blank checking);
    if (blank checking)
        return FM get min max aux(field, t, sid, tti);
    std::pair<T,T> min max aux(field->data[0], field->data[0]);
    const T* d = field->data + i;
    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;
}
```

```
// Emacs mode * c++ *
#endif
```
public:
  FM_classic_morph_field_2_layout(const FM_ptr<FM_structured_mesh<B,D>>& m, const T* vals) const
  { }

virtual int
at_cell(const FM_cell* c, T* vals) const  
{  
  if (m_indices == 1)
    vals[0] = *vals;  
  for (FM u32 i = 0; i < n_indices; i++)
  {  
    T* dst = &(*vals)[previous_size];  
    vals[i] = data[indices[i]];  
  }  
  return FM_OK;  
}

virtual int
at_cell(const FM_cell* cm, T* vals) const
{  
  FM_ptr<FM_structured_mesh<B,D> > m = cm;  
  for (FM u32 i = 0; i < n_indices; i++)
  {  
    T* dst = &(*vals)[previous_size];  
    vals[i] = data[indices[i]];  
  }  
  return FM_OK;  
}

return FM_OK;  
}

virtual int
at_cell(const FM_cell* cm, T* vals) const
{  
  FM_ptr<FM_structured_mesh<B,D> > m = cm;  
  for (FM u32 i = 0; i < n_indices; i++)
  {  
    T* dst = &(*vals)[previous_size];  
    vals[i] = data[indices[i]];  
  }  
  return FM_OK;  
}

return FM_OK;  
}

/**
   * Copyright 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  

#ifndef FMCURVILINEAR_MESH_H

// 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奇葩 needs to be called by the derived classes that declare at_cell so this routine...
/// has the option of accessing vertex coordinates as part of the
/// initialization.
void curvilinear_mesh_initialize()
{
    FM_vector<FM, FM_u32> initial_location = dimensions;
    initial_location /= 2;
    initial_locations.push_back(initial_location);
}

public:
FM_curvilinear_mesh(const FM_vector<3, FM_u32>& dimensions,
    FM_properties_cache* pc = 0,
    bounding_box_valid=false)
{
    std::cout << std::endl;
    virtual int size_t::operator() const
    { return 0; }
    std::ostream& str(std::ostream& o) const
    { return o << "FM curvilinear mesh<3,3>"; }
}

FM curvilinear_mesh<3,3>(const FM vector<3, FM u32>& dimensions,
    FM_properties_cache* pc = 0,
    bounding_box_valid=false)
{
    virtual std::ostream& str(std::ostream& o) const
    { return o << "FM curvilinear mesh<3,3>"; }
    FM_vector<FM, FM_u32> initial_location;
    FM_vector<3, FM u32> initial_locations;
    initial_locations.push_back(FM_vector<3, FM u32>(i, diml_050, dim2_050));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, i));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, dim2_050));
    return FM_curvilinear_mesh<3,3>(dimensions, pc);
}

FM_vector<FM, FM_u32> initial_location;
FM_vector<3, FM u32> initial_locations;
mutable bool bounding box valid;
mutable std::pair<FM vector<m, FM_coord>, FM vector<m, FM_coord> > boundingbox;
// assume

FM structured_mesh<3,3>(dimensions, pc, bounding_box_valid=false)
{
    std::vector<FM vector<3, FM u32> > initial locations;
    mutable bool bounding box -valid;
    mutable std::pair<FM vector<m, FM_coord>, FM vector<m, FM_coord> > boundingbox;
    // assume
}

FM structured_mesh<3,3>(dimensions, pc),

FM properties cache* pc = 0;
bounding_box_valid=false)
{
    std::cout << std::endl;
    virtual int size_t::operator() const
    { return 0; }
    std::ostream& str(std::ostream& o) const
    { return o << "FM curvilinear mesh<3,3>"; }
}

FM curvilinear_mesh<3,3>(const FM_vector<3, FM_u32>& dimensions,
    FM_properties_cache* pc = 0,
    bounding_box_valid=false)
{
    virtual std::ostream& str(std::ostream& o) const
    { return o << "FM curvilinear mesh<3,3>"; }
    FM_vector<FM, FM_u32> initial_location;
    FM_vector<3, FM u32> initial_locations;
    initial_locations.push_back(FM_vector<3, FM u32>(i, diml_050, dim2_050));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, i));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, dim2_050));
    return FM_curvilinear_mesh<3,3>(dimensions, pc);
}

FM curvilinear_mesh<3,3>(const FM_vector<3, FM_u32>& dimensions,
    FM_properties_cache* pc = 0,
    bounding_box_valid=false)
{
    virtual std::ostream& str(std::ostream& o) const
    { return o << "FM curvilinear mesh<3,3>"; }
    FM_vector<FM, FM_u32> initial_location;
    FM_vector<3, FM u32> initial_locations;
    initial_locations.push_back(FM_vector<3, FM u32>(i, diml_050, dim2_050));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, i));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, dim2_050));
    return FM_curvilinear_mesh<3,3>(dimensions, pc);
}

FM curvilinear_mesh<3,3>(const FM_vector<3, FM_u32>& dimensions,
    FM_properties_cache* pc = 0,
    bounding_box_valid=false)
{
    virtual std::ostream& str(std::ostream& o) const
    { return o << "FM curvilinear mesh<3,3>"; }
    FM_vector<FM, FM_u32> initial_location;
    FM_vector<3, FM u32> initial_locations;
    initial_locations.push_back(FM_vector<3, FM u32>(i, diml_050, dim2_050));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, i));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, dim2_050));
    return FM_curvilinear_mesh<3,3>(dimensions, pc);
}

FM curvilinear_mesh<3,3>(const FM_vector<3, FM_u32>& dimensions,
    FM_properties_cache* pc = 0,
    bounding_box_valid=false)
{
    virtual std::ostream& str(std::ostream& o) const
    { return o << "FM curvilinear mesh<3,3>"; }
    FM_vector<FM, FM_u32> initial_location;
    FM_vector<3, FM u32> initial_locations;
    initial_locations.push_back(FM_vector<3, FM u32>(i, diml_050, dim2_050));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, i));
    initial_locations.push_back(FM_vector<3, FM u32>(dim0_050, diml_050, dim2_050));
    return FM_curvilinear_mesh<3,3>(dimensions, pc);
}
if (ctxt >locate verbosity > i)
ctxt >last cell = *c;
else
]

if (!suppressed step off mesh) {
]

next subtetrahedron face:
while (faces tested < 4) {
FM vector<3,FM coord> cv[8];
bool suppressed step off mesh = false;
FM u32 total faces tested = 0;
FM u32 face = 0;

if (initial >is subsimplex())
FM vector<3,FM u32> indices = initial >get indices();
assert(initial
res = FM POINT LOCATE WALKED OFF MESH;
if (ctxt >simplicial decomposition)
faces tested += i;
face = (face + i) & 3;

if (outside) {
if (orientation == 0)
FM orient(cv[FM subtetrahedron face[subid][face][0]],
FM_orient_name(orientation + 1) << std::endl;
}

if (total faces tested > total faces tested threshold)
FM nominated_flag = false;
total faces tested += 1;

std::cout << verbose prefix.strO << std::endl;

return hexahedral walk locate(p, *c, ctxt, c);

for (FM u32 i = 0; i < initial locations.size(); i++) {
FM structured 0 cell<3> v;
FM u32 axis;
if (indices[axis] == dimensions[axis] 2) {
break;
}

if (!initial locations priority queue.empty()) {
std::vector<FM vector<3,FM u32> >
for (axis = 0;
FM u32 axis;
PM u32 prev_subid = subid;
PM u32 prev_face = face;
subid = FM tetrahedron step[prev_subid][prev_face].subsimplex;
face = FM tetrahedron step[prev_subid][prev_face].subsimplex_face;
facets tested = 0;
suppressed step off mesh = false;

FM structured 0 cell<3> v;
FM u32 axis;
if (indices[axis] == dimensions[axis] 2) {
break;
}

if (outside) {
break;
}
next subtetrahedron face;
face = (face + i) & 3;
facets tested += 1;

if (!suppressed step off mesh) {
FM structuring_bounding_box_odd = PM
FM oriented_flag(orientation + 1);

FM u32 axis;
if (indices[axis] == dimensions[axis] 2) {
break;
}

if (outside) {
break;
}
next subtetrahedron face;
face = (face + i) & 3;
facets tested += 1;

if (!suppressed step off mesh) {
FM structuring_bounding_box_odd = PM
FM oriented_flag(orientation + 1);

FM u32 axis;
if (indices[axis] == dimensions[axis] 2) {
break;
}

if (outside) {
break;
}
private:
    std::vector<FM_vector<3, FM_coord>> 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<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<FM_coord>* const coordinates;
    const bool delete suppression;
    FM_curvilinear_mesh_T_layout(const FM_vector<FM_coord>* const coordinates, 
const bool delete suppression);

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// the coordinates for each vertex are contiguous.  
// template <int B, int D>
class FM_curvilinear_mesh_T_layout<B,D> : public FM_curvilinear_mesh<B,D>
public:
    const FM_vector<FM_coord>* const coordinates;
    const bool delete suppression;
    FM_curvilinear_mesh_T_layout(const FM_vector<FM_coord>* const coordinates, 
const bool delete suppression);

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// the coordinates for each vertex are contiguous.
// template <int B, int D>
FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// the coordinates for each vertex are contiguous.
// template <int B, int D>
FM_curvilinear_mesh_T_layout<B,D>() : 
FM_curvilinear_mesh_T_layout<B,D>(dimensions, properties_cache, pc) {
    FM_vector<D, FM_coord>* cp coordinates;
    std::pair<FM_vector<3, FM_coord>, FM_vector<3, FM_coord>> bb;

    for (FM u64 i = 0; i < dimensions[0]; i++)
        for (FM u64 j = i; j < dimensions[l]; j++)
            for (FM u64 k = j; k < dimensions[2]; k++) {
                FM operator min max equals(bb, *cp);
                bb.second = bb;  
                bb.second = *cp;
                cp++;
            }
    
    return bounding box;

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// a pointer directly into the coordinates buffer, testing every vertex.
// template <int B, int D>
FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// a pointer directly into the coordinates buffer, testing every vertex.
// template <int B, int D>
std::pair<FM_vector<3, FM_coord>, FM_vector<3, FM_coord>>
get bounding box(const FM_vector<FM_coord>* = 0, 
                const FM_vector<FM_coord>* const coordinates = 0, 
const FM_vector<FM_coord>* const coordinates, 
const bool delete suppression);

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// a pointer directly into the coordinates buffer, testing every vertex.
// template <int B, int D>
std::pair<FM_vector<3, FM_coord>, FM_vector<3, FM_coord>>
get bounding box(const FM_vector<FM_coord>* = 0, 
                const FM_vector<FM_coord>* const coordinates = 0, 
const FM_vector<FM_coord>* const coordinates, 
const bool delete suppression);

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// a pointer directly into the coordinates buffer, testing every vertex.
// template <int B, int D>
std::pair<FM_vector<3, FM_coord>, FM_vector<3, FM_coord>>
get bounding box(const FM_vector<FM_coord>* = 0, 
                const FM_vector<FM_coord>* const coordinates = 0, 
const FM_vector<FM_coord>* const coordinates, 
const bool delete suppression);

/*
 * 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

 */

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// a pointer directly into the coordinates buffer, testing every vertex.
// template <int B, int D>
std::pair<FM_vector<3, FM_coord>, FM_vector<3, FM_coord>>
get bounding box(const FM_vector<FM_coord>* = 0, 
                const FM_vector<FM_coord>* const coordinates = 0, 
const FM_vector<FM_coord>* const coordinates, 
const bool delete suppression);

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// a pointer directly into the coordinates buffer, testing every vertex.
// template <int B, int D>
std::pair<FM_vector<3, FM_coord>, FM_vector<3, FM_coord>>
get bounding box(const FM_vector<FM_coord>* = 0, 
                const FM_vector<FM_coord>* const coordinates = 0, 
const FM_vector<FM_coord>* const coordinates, 
const bool delete suppression);

// FM_curvilinear_mesh_T_layout<3,3>::get bounding box works with
// a pointer directly into the coordinates buffer, testing every vertex.
template <int B, int D, typename R, typename S, typename T, typename F>
    struct FM binary derived field : public FM field<B,D,T>
    {
        //...
```cpp
class FMfield : public FMsharedobject with properties cache
protected:
public:
class FMfield interface:

template <int B, int D, typename T>

virtual int FMget min max(aux(const FM submesh id* sid) const

FM properties cache* pc) :

FM variable namesaux(property names, t, sid);

property names.insert("mesh property");
try {
    mesh property = mesh_property;
    catch (...) {

    if (mesh property == 0)
        return mesh property;
    return FM variable namesaux with properties cache::
        get aux(key, pass, t, sid);
}
}

virtual std::string FMget property names aux(property names, t, sid)
    { return std::set<std::string> aux(property names, t, sid);
    return FM variable names aux(property names, t, sid);
    }

return FM variable names aux(property names aux property names, t, sid);

template <typename T, typename S>

FM 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 == "field type name")
        return new FM simple value<FM context>(t, val);
    else if (key == "mesh property")
        return new FM simple value<FM variable names aux(property names aux property names, t, sid);
    return FM get aux(key, pass, t, sid);
    }

std::string FMget aux(const FM variable names aux(property names aux property names, t, sid);
    { property names insert("node type");
        property names insert("field type name");
        property names insert("mesh property");
        std::set<std::string> mesh property names = mesh_property_names(t, sid);
        std::set<std::string> property names aux(property names aux(property names, t, sid);
        return FM variable names aux(property names aux(property names aux property names, t, sid);
        };
}
```
#ifndef FM_FIELD_INTERFACE_H
#define FM_FIELD_INTERFACE_H

/*
 * 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 "FMsharedobject.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_func : public std::unary function<T,T>
{
public:
int operatorO(const T& t, T* res) const
{
*res = -t;
return FM_OK;
}
};

template <typename T>
class FM_plus_func : public std::binary function<T,T,T>
{
public:
int operatorO(const T& lhs, const T& rhs, T* res) const
{
*res = lhs + rhs;
return FM_OK;
}
};

template <typename T>
class FM_minus_func : public std::binary function<T,T,T>
{
public:
int operatorO(const T& lhs, const T& rhs, T* res) const
{
*res = lhs - rhs;
return FM_OK;
}
};

template <typename T>
class FM_multiplies_func : public std::binary function<T,T,T>
{
public:
int operatorO(const T& lhs, const T& rhs, T* res) const
{
*res = lhs * rhs;
return FM_OK;
}
};

template <typename T>
class FM_divides_func : public std::binary function<T,T,T>
{
public:
int operatorO(const T& lhs, const T& rhs, T* res) const
{
*res = lhs / rhs;
return FM_OK;
}
};

template <typename S, typename T>
class FM_abs_func : public std::unary function<T,T>
{
public:
int operatorO(const T& t, T* res) const
{
*res = t > T(0) ? t : -t;
return FM_OK;
}
};

template <typename S, typename T>
class FM_max_func : public std::binary function<T,T,T>
{
public:
int operatorO(const T& lhs, const T& rhs, T* res) const
{
*res = lhs > rhs ? lhs : rhs;
return FM_OK;
}
};

template <typename T>
class FM_min_func : public std::binary function<T,T,T>
{
public:
int operatorO(const T& lhs, const T& rhs, T* res) const
{
*res = lhs < rhs ? lhs : rhs;
return FM_OK;
}
};

template <typename T>
class FM_dot_func : public std::binary function<FM_vector<N,T>,FM_vector<N,T>,T>
{
public:
int operatorO(const FM_vector<N,T>& lhs, const FM_vector<N,T>& rhs, T* res) const
{
*res = FMdot(lhs, rhs);
return FM_OK;
}
};

template <typename T>
class FM_cross_func : public std::binary function<FMvector<3,T>,FMvector<3,T>,FMvector<3,T> >
{
public:
int operatorO(const FMvector<3,T>& lhs, const FMvector<3,T>& rhs, FMvector<3,T>* res) const
{
*res = FMcross(lhs, rhs);
return FM_OK;
}
};

template <typename T>
class FM_mag_func : public std::unary function<FM_vector<N,T>,T>
{
public:
int operatorO(const FM_vector<N,T>& v, T* res) const
{
*res = FMmag(v);
return FM_OK;
}
};

template <typename T>
class FM_brackets_func : public std::unary function<FM_vector<3,T>,T>
{
public:
int operatorO(const FM_vector<3,T>& v, T* res) const
{
*res = v[index];
return FM_OK;
}
};

private:
const int index;
};

template <typename T>
class FM_slice_brackets_func : public std::unary function<FM_vector<M,T>,FM_vector<N,T> >
{
public:
int operatorO(const FM_vector<M,T>& v, int index) const
{
*res = v[index];
return FM_OK;
}
};

private:
const int index;
};

template <typename T>
class FM_slice_brackets_func : public std::unary function<FM_vector<M,T>,T>
{
public:
int operatorO(const FM_vector<M,T>& v, int index) const
{
*res = v[index];
return FM_OK;
}
};

private:
const int index;
};

}
";'>; FM slicebrackets fun(" << i << ");
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;
return FM_OK;
}

private:
const int index;
};

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;
}

private:
const int index;
};

template <typename T>
class 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()'

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) {
}

FM component (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.
*/

template <typename T>
class FM swap endian fun : public std::unary function<float, float>
{

private:
const float first;
const float second;
};

*/
```cpp
// Define the type
#ifndef FM_INTERPOLATE_H
#define FM_INTERPOLATE_H

// Name: FM_interpolate.h

// Written by: Patrick Moran

#include <vector>
#include "FM_linear_interpolate.h"
#include "FM_vector.h"
#include "FM_cell.h"
#include "FM_returns.h"

// FM linear interpolate(const FM base<B>& b, int T vals[], T* val)
int FM linear interpolate(const FM vector<B, FMcoord>& f, T vals[], T* val) {
    for (int i = B - 1; i >= 0; i--) {
        if (f[i] FMcoord(0)) continue;
        vals[i] = f[i] * (vals[i + n] - vals[i]);
    }
    return 0;
    return FMOK;
}

// FM vector<B,FMcoord> f;

// template <int B, typename T> int FM fread(FILE* fp, T* dat, sizet n items, bool swap endian, bool fortran, const FM ptr<FMstructured B cell<B>>& sc, const FM_ptr<FMstructured B cell<B>>& extra, const FM_ptr<FMstructured B cell<B>>& extra2, bool extra, const FM_ptr<FMstructured B cell<B>>& extra3, bool extra3)
int FM fread(FILE* fp, T* dat, sizet n items, bool swap endian) {
    if (fortran) {
        // Pack struct stat
        struct stat sbuf;
        if (stat(fp, &sbuf)) return FM I0 ERROR;
    }
    // Check byte order
    if (swap endian) {
        if (extra && n bytes fortran > n bytes expected) {
            // Extra information
            if (n bytes fortran < n bytes expected)
                if (res != FM OK) return res;
            res = FM fread(fp, &(*extra)[0], n extra, swap endian);
            extra_ >resize(n extra);
            size t n extra = (n bytes fortran n bytes expected)
        }
        // Swap endian
        for (size t i = 0; i < fread res; i++) {
            FM swap endian fun<T> swap endian fun;
            return FM I0 ERROR;
        }
    }
    // Read data
    if (fortran) {
        size t start = ftell(fp);
        FM u32 nbytesfortran;
        FMfread(fp, &nbytesfortran, i, swap endian);
        if (nbytesfortran i < n bytes expected)
            return FM fread(res, n items, swap endian);
        size t fread res fread(dat, sizeof(T), n items, fp);
        if (swap endian) {
            // Swap endian
            for (size t i = 0; i < fread res; i++) {
                return FM I0 ERROR;
            }
        }
    }
    // Read data
    if (extra && extra)
        if (extra) {
            // Extra information
            if (extra && extra)
                if (extra) return res;
            if (!extra) return FMOK;
            // Extra information
            if (extra && extra) {
                // Extra information
                if (extra) return res;
            }
        }
    // Read data
    for (size t i = 0; i < n items; i++) {
        FM little endian hardware();
        if (extra && extra) {
            // Extra information
            if (extra) return res;
        }
    }
    return FMOK;
}

// FM_linear_interpolate(const FM base<B>& b, int T vals[], T* val)
int FM_linear_interpolate(const FM vector<B, FMcoord>& f, T vals[], T* val) {
    if (sizeof(argument type) sizeof(resulttype)) {
        // Pack struct stat
        struct stat sbuf;
        if (stat(fp, &sbuf)) return FM I0 ERROR;
    }
    // Check byte order
    if (swap endian) {
        if (extra && n bytes fortran > n bytes expected) {
            // Extra information
            if (n bytes fortran < n bytes expected)
                if (res != FM OK) return res;
            res = FM fread(fp, &(*extra)[0], n extra, swap endian);
            extra_ >resize(n extra);
            size t n extra = (n bytes fortran n bytes expected)
        }
        // Swap endian
        for (size t i = 0; i < fread res; i++) {
            FM swap endian fun<T> swap endian fun;
            return FM I0 ERROR;
        }
    }
    // Read data
    if (fortran) {
        size t start = ftell(fp);
        FM u32 nbytesfortran;
        FMfread(fp, &nbytesfortran, i, swap endian);
        if (nbytesfortran i < n bytes expected)
            return FM fread(res, n items, swap endian);
        size t fread res fread(dat, sizeof(argument type),.fp);
        if (swap endian) {
            // Swap endian
            for (size t i = 0; i < fread res; i++) {
                return FM I0 ERROR;
            }
        }
    }
    // Read data
    if (extra && extra) {
        // Extra information
        if (extra) return res;
    }
    // Read data
    for (size t i = 0; i < n items; i++) {
        FM little endian hardware();
        if (extra && extra) {
            // Extra information
            if (extra) return res;
        }
    }
    return FMOK;
}

// FM_linear_interpolate(const FM base<B>& b, int T vals[], T* val)
int FM_linear_interpolate(const FM vector<B, FMcoord>& f, T vals[], T* val) {
    if (sizeof(argument type) sizeof(resulttype)) {
        // Pack struct stat
        struct stat sbuf;
        if (stat(fp, &sbuf)) return FM I0 ERROR;
    }
    // Check byte order
    if (swap endian) {
        if (extra && n bytes fortran > n bytes expected) {
            // Extra information
            if (n bytes fortran < n bytes expected)
                if (res != FM OK) return res;
            res = FM fread(fp, &(*extra)[0], n extra, swap endian);
            extra_ >resize(n extra);
            size t n extra = (n bytes fortran n bytes expected)
        }
        // Swap endian
        for (size t i = 0; i < fread res; i++) {
            FM swap endian fun<T> swap endian fun;
            return FM I0 ERROR;
        }
    }
    // Read data
    if (fortran) {
        size t start = ftell(fp);
        FM u32 nbytesfortran;
        FMfread(fp, &nbytesfortran, i, swap endian);
        if (nbytesfortran i < n bytes expected)
            return FM fread(res, n items, swap endian);
        size t fread res fread(dat, sizeof(argument type), fp);
        if (swap endian) {
            // Swap endian
            for (size t i = 0; i < fread res; i++) {
                return FM I0 ERROR;
            }
        }
    }
    // Read data
    if (extra && extra) {
        // Extra information
        if (extra) return res;
    }
    // Read data
    for (size t i = 0; i < n items; i++) {
        FM little endian hardware();
        if (extra && extra) {
            // Extra information
            if (extra) return res;
        }
    }
    return FMOK;
}

// FM_linear_interpolate(const FM base<B>& b, int T vals[], T* val)
int FM_linear_interpolate(const FM vector<B, FMcoord>& f, T vals[], T* val) {
    if (sizeof(argument type) sizeof(resulttype)) {
        // Pack struct stat
        struct stat sbuf;
        if (stat(fp, &sbuf)) return FM I0 ERROR;
    }
    // Check byte order
    if (swap endian) {
        if (extra && n bytes fortran > n bytes expected) {
            // Extra information
            if (n bytes fortran < n bytes expected)
                if (res != FM OK) return res;
            res = FM fread(fp, &(*extra)[0], n extra, swap endian);
            extra_ >resize(n extra);
            size t n extra = (n bytes fortran n bytes expected)
        }
        // Swap endian
        for (size t i = 0; i < fread res; i++) {
            FM swap endian fun<T> swap endian fun;
            return FM I0 ERROR;
        }
    }
    // Read data
    if (fortran) {
        size t start = ftell(fp);
        FM u32 nbytesfortran;
        FMfread(fp, &nbytesfortran, i, swap endian);
        if (nbytesfortran i < n bytes expected)
            return FM fread(res, n items, swap endian);
        size t fread res fread(dat, sizeof(argument type), fp);
        if (swap endian) {
            // Swap endian
            for (size t i = 0; i < fread res; i++) {
                return FM I0 ERROR;
            }
        }
    }
    // Read data
    if (extra && extra) {
        // Extra information
        if (extra) return res;
    }
    // Read data
    for (size t i = 0; i < n items; i++) {
        FM little endian hardware();
        if (extra && extra) {
            // Extra information
            if (extra) return res;
        }
    }
    return FMOK;
}

// FM_linear_interpolate(const FM base<B>& b, int T vals[], T* val)
int FM_linear_interpolate(const FM vector<B, FMcoord>& f, T vals[], T* val) {
    if (sizeof(argument type) sizeof(resulttype)) {
        // Pack struct stat
        struct stat sbuf;
        if (stat(fp, &sbuf)) return FM I0 ERROR;
    }
    // Check byte order
    if (swap endian) {
        if (extra && n bytes fortran > n bytes expected) {
            // Extra information
            if (n bytes fortran < n bytes expected)
                if (res != FM OK) return res;
            res = FM fread(fp, &(*extra)[0], n extra, swap endian);
            extra_ >resize(n extra);
            size t n extra = (n bytes fortran n bytes expected)
        }
        // Swap endian
        for (size t i = 0; i < fread res; i++) {
            FM swap endian fun<T> swap endian fun;
            return FM I0 ERROR;
        }
    }
    // Read data
    if (fortran) {
        size t start = ftell(fp);
        FM u32 nbytesfortran;
        FMfread(fp, &nbytesfortran, i, swap endian);
        if (nbytesfortran i < n bytes expected)
            return FM fread(res, n items, swap endian);
        size t fread res fread(dat, sizeof(argument type), fp);
        if (swap endian) {
            // Swap endian
            for (size t i = 0; i < fread res; i++) {
                return FM I0 ERROR;
            }
        }
    }
    // Read data
    if (extra && extra) {
        // Extra information
        if (extra) return res;
    }
    // Read data
    for (size t i = 0; i < n items; i++) {
        FM little endian hardware();
        if (extra && extra) {
            // Extra information
            if (extra) return res;
        }
    }
    return FMOK;
}
```
```c
int FM fread transpose fun(FILE* fps[], int n_items, bool swap_endian, bool fortran) {
    template <int N, typename T>
    while (n_remaining > 0) {
        int res;
        size_t n_remaining = n_items;
        
        for (i = 0; i < N; i++) {
            const FM u32 N mAT = 1024;
            std::vector<argument type> components[N];
            
            size_t n_bytes fortran;
            size_t start = ftell(fps[0]);
            
            typedef typename F::result type result type;
            typedef typename F::argument type argument type;
            
            if (fortran)
                n_bytes fortran = n_items * N * sizeof(argument type);
            else
                n_bytes fortran = n_items * sizeof(T);
            
            if (n_bytes fortran < N * sizeof(T) * n_items)
                FM fread(fps[0], &n_bytes fortran, i, swap endian);  
            
            size_t n_remaining = n_items - N * sizeof(T);
            t* dst = reinterpret cast<T*>(dat);
            
            size_t n_remaining = n_bytes fortran;
            while (n_remaining > 0) {
                size_t n_to_read = (n_remaining > N mAT) ? N mAT : n_remaining;
                if (res != FM OK)
                    return res;
                
                res = FM fread(fps[i], &components[i][0], n_to_read, swap endian);
                if (res != FM OK)
                    return res;
                
                for (i = 0; i < N; i++) {
                    size_t n_remaining = n_items;
                    t* dat = reinterpret cast<T*>(dat);
                    
                    size_t n_remaining = n_to_read;
                    while (n_remaining > 0) {
                        size_t n_to_read = (n_remaining > N mAT) ? N mAT : n_remaining;
                        if (n_to_read > 0)
                            for (i = 0; i < N; i++) {
                                res = FM fread(fps[i], &components[i][j], n_to_read, swap_endian);
                                if (res != FM OK)
                                    return res;
                                
                                for (j = 0; j < n_to_read; j++) {
                                    if (res != FM OK)
                                        return res;
                                    
                                    dst[j] = reinterpret cast<result type*>(dat)[j];
                                    dat += sizeof(T);
                                }
                            }
                    }
                }
            }
        }
    }
    return FM OK;
}

template <typename T>
int FM fwrite transpose supp(FILE* fp, const FM vector<N,T>* dat, int n_items, bool swap_endian, bool fortran) {
    template <typename T>
    size_t i, j;
    std::vector<T> components[N];
    
    size_t n_remaining = n_items;
    t* dat = reinterpret cast<T*>(dat);
    
    size_t n_bytes_fortran;
    size_t start = ftell(fp);  
    
    typedef typename F::result type result type;
    typedef typename F::argument type argument type;
    
    if (fortran)
        n_bytes_fortran = n_items * N * sizeof(argument type);
    else
        n_bytes_fortran = n_items * sizeof(T);
    
    if (n_bytes_fortran < N * sizeof(T) * n_items)
        FM fwrite(fp, &n_bytes_fortran, i, swap endian);
    
    size_t n_remaining = n_items - N * sizeof(T);
    t* dst = reinterpret cast<T*>(dat);
    
    size_t n_remaining = n_bytes_fortran;
    while (n_remaining > 0) {
        size_t n_to_read = (n_remaining > N mAT) ? N mAT : n_remaining;
        if (res != FM OK)
            return res;
        
        res = FM fwrite(fp, &components[i][0], n_to_read, swap endian);
        if (res != FM OK)
            return res;
        
        for (i = 0; i < N; i++) {
            size_t n_remaining = n_to_read;
            while (n_remaining > 0) {
                size_t n_to_read = (n_remaining > N mAT) ? N mAT : n_remaining;
                if (n_to_read > 0)
                    for (i = 0; i < n_to_read; i++) {
                        if (res != FM OK)
                            return res;
                        
                        dst[i] = reinterpret cast<result type*>(dat)[i];
                        dat += sizeof(T);
                    }
            }
        }
    }
    return FM OK;
}
```

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.
/* 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 "FMsharedobject.h"
#include "FMcell.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 attr rs;

  std::ostream& operator<<(std::ostream& o, const FM iter attr rs& ia)
  {
    o << "NU
for (FM u32 i = 0; i < ia.sizeO; 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(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 u32, axis_end_iter_attr), axis(a), index(i) {
    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 u32, axis_stride_iter_attr), axis(a), stride(s) {
    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 u32, simplicial_decomposition_iter_attr), simplicial_decomposition(sd) {
    return o << "FM simplicial_decomposition_iter_attr(" << simplicial_decomposition << ");";
}
};
class FM iter_impl
public:
virtual "FM iter_impl();" {};
virtual FM_iter_impl* copy() const { o << "FM iter_impl();";
    return o;
}
};
class FM_iter
public:
FM_iter() : impl(0), cell(0) {}
FM iter(FM iter_impl* i) : impl(i), cell(i->dereference()) {}
FM iter(const FM_iter& iter) :
    impl(iter.impl->copy()), cell(iter.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 FMCell* operator++()
return cell = impl->advance();
void operator++(int) { (void) operator++; }
inline const FMCell* operator*() const { return cell; }
inline bool done() const { return cell == 0; }
fReader bool operator>(const FM_iter l, const FM_iter r)
if (lcell.cell == 0 || rcell.cell == 0)
    return lcell.cell == rcell.cell;
else
    return *lcell.cell == *rcell.cell;
friend bool operator<(const FM_iter l, const FM_iter r)
return !operator>(r, l);

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

*/

* Copyright © 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.
* /
template<typename T>

T minor3 const FM vector<N,F vector<N, T> >& in, int row, int col) {
    return in[0][0] * minor0 + in[2][2] * minor2 + in[3][3] * minor3;
}

template<typename T>

T minorl const FM vector<N,F vector<N, T> >& in, int row, int col) {
    return in[0][0] * minor0 + in[2][2] * minor2 + in[3][3] * minor3;
}

template<typename T>

T minor0 const FM vector<N,F vector<N, T> >& in, int row, int col) {
    return in[0][0] * minor0 + in[2][2] * minor2 + in[3][3] * minor3;
}

template<typename T>

T FMdet(const FM vector<N,F vector<N, T> >& in) {
    T det = 0;
    for (int row = 0; row < N; row++) {
        for (int col = row; col < N; col++) {
            T cofactor = (row == col) ? 1 : (row < col) ? cofactor : -cofactor;
            T sum = in[row][0] * cofactor;
            for (int m = 0; m < N; m++) {
                T tmp = in[m][col];
                for (int n = 0; n < N; n++) {
                    T tmp2 = in[m][n] * cofactor;
                    sum += tmp2;
                }
                tmp = sum;
                sum = 0;
                for (int n = 0; n < N; n++) {
                    T tmp3 = in[n][col] * cofactor;
                    sum += tmp3;
                }
                tmp = sum;
                sum = 0;
                for (int n = 0; n < N; n++) {
                    T tmp4 = in[n][m] * cofactor;
                    sum += tmp4;
                }
                tmp = sum;
                sum = det + tmp;
            }
            det += tmp;
        }
    }
    return det;
}

template<typename T>

T FM transpose(const FM vector<3,F vector<3, T> >& in) {
    F M = trap;
    for (int m = 0; m < M; m++) {
        for (int n = 0; n < N; n++) {
            T tmp = in[m][n] * trap;
            tmp[m][n] = tmp;
        }
    }
    return FM transpose<3,T>(in);
}

template<typename T>

T FM transpose(const FM vector<4,F vector<4, T> >& in) {
    F M = trap;
    for (int m = 0; m < M; m++) {
        for (int n = 0; n < N; n++) {
            T tmp = in[m][n] * trap;
            tmp[m][n] = tmp;
        }
    }
    return FM transpose<4,T>(in);
}

template<typename T>

T FM minor(const FM vector<N,F vector<N, T> >& in, int row, int col) {
    T minor = 0;
    for (int m = 0; m < N; m++) {
        for (int n = 0; n < N; n++) {
            T tmp = in[m][n] * trap;
            tmp[m][n] = tmp;
        }
    }
    return FM minor<N,T>(in, row, col);
}

template<typename T>

T FM det(const FM vector<N,F vector<N, T> >& in) {
    T det = 0;
    for (int row = 0; row < N; row++) {
        for (int col = row; col < N; col++) {
            T cofactor = (row == col) ? 1 : (row < col) ? cofactor : -cofactor;
            T sum = in[row][0] * cofactor;
            for (int m = 0; m < N; m++) {
                T tmp = in[m][col];
                for (int n = 0; n < N; n++) {
                    T tmp2 = in[m][n] * cofactor;
                    sum += tmp2;
                }
                tmp = sum;
                sum = 0;
                for (int n = 0; n < N; n++) {
                    T tmp3 = in[n][col] * cofactor;
                    sum += tmp3;
                }
                tmp = sum;
                sum = 0;
                for (int n = 0; n < N; n++) {
                    T tmp4 = in[n][m] * cofactor;
                    sum += tmp4;
                }
                tmp = sum;
                sum = det + tmp;
            }
            det += tmp;
        }
    }
    return det;
}
class FM mesh : public FM shared object with properties cache
class FM mesh : \[template \langle B, D \rangle \]
};

public:
{  

class FM mesh : public FM shared object with properties cache

#include "FMiter.h"
#include "FMsharedobjectwithpropertiescache.h"

#define FM_MESHH

#ifndef FM_MESHH
//

• WRITTEN BY:
• NAME: FM mesh.h

public FM mesh ,

{


public FM mesh ,

{  


#endif

• PMEMORI*

emacs mode * c++ *
virtual FM_ptr<FM_share_object>&
get_aux(const std::string &key, FM &pass, 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_share_object> > lo_values(D);
    std::vector<FM_ptr<FM_share_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<FM_tuple_value>(lo_values, hi_values);
  }
  return FM_mesh::get_aux(key, pass, t, sid);
}

virtual std::set<std::string>
get_property_names(FM_aux &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(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

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
// FMorient template:
const FMorient (const FMvector<3, T>& a, const FMvector<3, T>& b, const FMvector<3, T>& c, const FMvector<3, T>& d)
{ 
    typedef double T;
    int res = FMsign(FMdot(d-a, FMcross(b-a, c-a)));
    for (int i = 0; i < 4; i++)
    { 
        if (FMabs(results[i]) > FMabs(results[largest result]))
        { 
            largest result = i;
        }
        std::cerr << triangle names[i] << " orient result: " << results[i] << std::endl;
        std::cerr << "largest magnitude result " << results[largest result] << std::endl;
    }
    return res;
}

template <>
const FMorient (const FMvector<2, T>& a, const FMvector<2, T>& b, const FMvector<2, T>& c) 
{ 
    // Compute orientation of a with respect to quadrilateral abcd
    // by treating abcd as two triangles: abc and 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.
    // 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,
    // return the largest magnitude result, based on triangles abd, abc,
    // bcd, and acd.
    // Compute orientation of a with respect to quadrilateral abcd
    // by treating abcd as two triangles: abc and 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.
    // 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,
    // return the largest magnitude result, based on triangles abd, abc,
    // bcd, and acd.
    // Compute orientation of a with respect to quadrilateral abcd
    // by treating abcd as two triangles: abc and 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.
    // 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,
    // return the largest magnitude result, based on triangles abd, abc,
    // bcd, and acd.
    // Compute orientation of a with respect to quadrilateral abcd
    // by treating abcd as two triangles: abc and 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.
    // 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,
    // return the largest magnitude result, based on triangles abd, abc,
```cpp
public:
    class FMostringstream : public std::ostrstream
    {
        // FM ostringstream provides a workaround for systems
        // that do not have std::ostringstream yet, using the
        // deprecated std::ostream. We basically define a
        // new class (FM ostringstream) derived from std::ostream,
        // with a redefined str() method that handles string termination
        // and memory management properly.
        #ifndef FM NO STRINGSTREAM
        typedef std::ostream FM ostringstream;
        #endif
        #ifndef FM NO STRINGSTREAM
        #if defined(GNUC)
            #define FMOSTRINGSTREAMH
            #ifndef FMOSTRINGSTREAMH
            #include <strstream>
            #endif
            #endif
        #else
            typedef std::ostream FM ostringstream;
        #endif
        #endif

        public:
            std::string str()
        {
            // 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 other reasons because we do
            // not know how it was allocated) so we "unfreeze" to transfer
            // ownership back to ostrstream.
            const char* res = std::ostrstream::str();
            freeze(false);
            return std::string(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.
 */
```cpp
#ifndef FMPRODUCTMESHH
#define FMPRODUCTMESHH

/*
• NAME: FM product mesh.h
• WRITTEN BY:
  Patrick Moran
gmoran@nas.nasa.gov
*/

#include "FMstructured 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<l,l> > > axes;

  FM product mesh(const std::vector<FM ptr<FM structured mesh<l,l> > >& a,
                  FM properties cache* pc = 0) :
    FM structured mesh<B,m>(pc),
    axes(a)
  {
    if (axes.size() != size_t(B)) {
      FM ostringstream err;
      err << "FM product mesh<" << B << "," << m << ">
            ::FM product mesh: expect " << B << " axes,
got " << axes.size();
      throw std::logic_error(err.str());
    }

    FM vector<l,FM u32> dimension;
    for (int i = 0; i < B; i++)
      dimension[i] = axes[i].get base dimensions();
    init(dimensions);
  }

protected:
  FM product mesh(const FM vector<B,FM u32>& d, FM properties cache* pc = 0) :
    FM structured mesh<B,m>(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 << "," << m << ">
  }

  virtual int at cell(const FM cell* c, std::vector<FM vector<m,FM coord> >* vals) const
  {
    const FM structured cell<B>* sc =
      dynamic cast<const FM structured cell<B>*>(c);
    FM vector<l,FM coord> coordinate;
    if (sc->is subsimplex()){
      for (int i = 0; i < D; i++)
        vals[i][j] = FM coord(0);
    }
  }

virtual int at cell(const FM cell* c, FM vector<m,FM coord>* vals) const
  {
    const FM structured cell<B>* sc =
      dynamic cast<const FM structured cell<B>*>(c);
    FM vector<l,FM coord> coordinate;
    if (sc->is subsimplex()){
      for (int i = 0; i < D; i++)
        vals[i][j] = FM coord(0);
    }
  }

};
#endif FMPRODUCTMESHH
```

This text contains a template class `FM product mesh` which inherits from `FM structured mesh`. It is used to create a product mesh by multiplying simpler mesh structures. The class includes methods for constructor arguments, cell queries, and coordinate extraction at the cell level. The template parameters `B` and `D` specify the number of bases and dimensions, respectively. The `init` function initializes the dimensions of the product mesh, and the `at cell` methods are used to query cell properties.
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.
typedef FM regular interval H
#ifndef FM REGULAR INTERVAL H
# define FM REGULAR INTERVAL H

// WRITTEN BY:
// NAME: FM regular_interval.h

class FM regular interval : public FM structured mesh<l,l>
{

public:
const FM coord origin;
const FM coord spacing;

FM regular_interval(FM u32 d,
const FM coord spacing;
const FM coord origin;
FM properties cache* pc = 0) :

FM structured mesh<d,d>(d, pc), origin(o), spacing(s) {
}

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

virtual int
at(const FM cell* c, std::vector<FM vector<l,FM coord> >* vals) const
{
int res = (FM coord(0) <= (*b)[0] &&
if (sc)
res = base to cell(*b, sc);
if (_es != FM OK) return res;
res = base to cell(*b, sc);
if (_es != FM OK) return res;

for (FM u32 i = 0; i < n_indices; i++)
c_structured mesh vertex indices(this, &n_indices, indices);
FM u64 indices[2];
FM u32 n_indices;
FM u64 indices[2];
FM u32 n_indices;
for (FM u32 i = 0; i < n_indices; i++)
vals[i] = origin + FM coord(indices[i]) * spacing;
return FM_OK;
}

virtual std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >
get bounding box(const FM time<FM u32>* = O, const FM _ubmesh id* = 0) const
{
return std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >

return res;
return res;
return res;
}

virtual std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >
get bounding box(const FM time<FM u32>* = O, const FM _ubmesh id* = 0) const
{
return std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >

}

phys to base(const FM phys<l>& p, FM context*, FM base<l>* b,
virtual int
at
virtual int
at
virtual std::ostream& str(std::ostream& o) const

return std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >

return std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >

return res;
return res;
return res;

}

}

#endif

}}}
```c++
const int FM_OK = 0;
const int FM_OUT_OF_BOUNDS = 1;
const int FM_RANGED_DATA = 2;
const int FM_POINT_LOCATION_FAILED = 3;
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;
```
```c++
// FM_ptr is a "smart pointer" for pointing at FM_sharedobject's.
template <typename S>
FM_ptr(const FM_ptr<const T>& rhs) :
ptr(static_cast<const T*>(rhs))
{
if (ptr) {
    ptr->increment_reference();
}
set(static_cast<const T*>(rhs));
return *this;
}
```
```c++
template <typename S>
const FM_ptr<T>& operator (const FM_ptr<const S>& rhs)
{
    if (static_cast<const S*>(rhs)) {
        T* t = static_cast<T*>(rhs);
        if (t) {
            FM_ptr<T>() = std::unique_ptr<T>(t);
        }
    }
    return *this;
}
```
```c++
inline const T* operator () const { return ptr; }
```
/* 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.
 */

#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 : new FM_properties_cache()) {
virtual FM_ptr<FM_shared_object> get(const std::string& key,
const FMtime<FMu32>* 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 = getaux(key, 0, t, sid);
// 3. second pass: opportunity to query composed classes
property = getaux(property, t, sid);
if (!property)
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*>(properties_cache)->set(key, property, t, sid);
return property;
}

virtual void set(const std::string& key, const FM_shared_object* property,
const FMtime<FMu32>* t 0, const FM submesh id* sid 0) const
{
const_cast<FM_properties_cache*>(properties_cache)->set(key, property, t, sid);
}

virtual std::set<std::string>
get_property_namesaux(std::set<std::string>& property names,
const FMtime<FMu32>* t,
const FM submesh id* sid) const
{
property names = properties_cache->get_property_namesaux(property names, t, sid);
return FM_shared_object::get_property_namesaux(property names, t, sid);
}

protected:
FM_ptr<FM_properties_cache> properties_cache; 
};

#endif
The canonical vertex numbering for a structured hexahedron. The vertex indices in the tables below are in terms of this hexahedron numbering.

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

```
FM u32* FM structured hexahedron faces[] = {
    FM u32 FM hsf3[] = {
        FM u32 FM hsf3 9[] = {i, 2, 4, 7},
        FM u32 FM hsf3 8[] = {2, 4, 7, 6},
        FM u32 FM hsf3 7[] = {i, 7, 4, 5},
        FM u32 FM hsf3 6[] = {i, 2, 7, 3},
        FM u32 FM hsf3 5[] = {i, 4, 2, 0},
        FM u32 FM hsf3 3[] = {0, 5, 6, 4},
        FM u32 FM hsf3 2[] = {0, 6, 3, 2},
        FM u32 FM hsf3 i[] = {0, 3, 5, i},
        FM u32 FM hsf3 0[] = {3, 6, 5, 7},
    },
    FM u32* FM hsf2[] = {
        FM u32 FM hf2[] = {
            FM u32 FM hsf2 19[] = {2, 7, 4},
            FM u32 FM hsf2 16[] = {i, 2, 4},
            FM u32 FM hsf2 15[] = {3, 5, 6},
            FM u32 FM hsf2 13[] = {0, 3, 6},
            FM u32 FM hsf2 12[] = {0, 5, 3},
            FM u32 FM hsf2 10[] = {0, 6, 4},
            FM u32 FM hsf2 9[] = {0, 6, 4},
            FM u32 FM hsf2 8[] = {0, 2, 6},
            FM u32 FM hsf2 7[] = {0, 2, 6},
            FM u32 FM hsf2 6[] = {0, 4, 5},
            FM u32 FM hsf2 5[] = {0, 4, 5},
            FM u32 FM hsf2 4[] = {2, 3, 7},
            FM u32 FM hsf2 3[] = {0, 3, 2},
            FM u32 FM hsf2 2[] = {0, 3, 2},
            FM u32 FM hsf2 1[] = {0, 3, 2},
            FM u32 FM hsf2 0[] = {0, 2, 4},
        },
        FM u32 FM hsf0[] = {FM hsf0 0[]},
    },
};
```

### Helper routines for `FM structured mesh.h`: Most routines
template <int B>
void FM structured mesh init(FM structured mesh<B,D>* sm, const std::vector<FM vector<B,FM u32>>& d) {
  for (i = 0; i < B; i++) {
    std::cout << "alignments[" << i << "]: ";
    for (j = 0; j < B; j++) {
      std::cout << "alignments[" << i << "][" << j << "]: ";
    }
  }
}

// The 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, int D>
// class FM structured mesh;
//
// template <int B, int D>
// FM u64 FM structured mesh get card(FM u32 base dimensionality, const FM vector<B,FM u32> indices, FM u32 structdim) {
//  FM u64 card = 0;
//  for (i = 0; i < B; i++) {
//    card *= indices[i];
//  }
//  return card;
//}

FM u64 index = sc->get_index(B - 1);
for (int i = B - 2; i >= 0; i--) {
  if (index >>= i + 1) {
    FM u32 d = index + i;
    for (int j = 0; j < D; j++) {
      index += alignment_dimensions[d][j][i] + sc->get_index(i);
    }
  }
}

// template <int B, int D>
// FM ptr<FM cell> FM structured mesh enum to cell(const std::vector<FM vector<B,FM u32>>& d) {
//  FM ptr<FM cell> res = new FM structured cell<B>(d);
//  for (a = 0; a < D; a++) {
//    std::cout << "cell offsets: ";
//    for (j = 0; j < B; j++) {
//      std::cout << "cube offsets[" << j << "]: ";
//    }
//  }
//  return res;
//}

FM u64 index = sc->get_index(B - 1);
for (int i = B - 2; i >= 0; i--) {
  for (int j = 0; j < D; j++) {
    if (index >>= i + 1) {
      FM u32 d = index + alignment_dimensions[d][j][i];
      index += alignment_cards(d)[i];
    }
  }
}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// FM u64 FM structured mesh get card(FM u32 base dimensionality, const FM vector<B,FM u32> indices, FM u32 structdim) {
//  FM u64 card = 0;
//  for (i = 0; i < B; i++) {
//    card *= indices[i];
//  }
//  return card;
//}

// template <int B>
// const FM time<FM u32>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// FM u64 FM structured mesh get card(FM u32 base dimensionality, const std::vector<FM vector<B,FM u32>>& d) {
//  FM u64 card = 0;
//  for (i = 0; i < B; i++) {
//    card *= d[i];
//  }
//  return card;
//}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}

// template <int B, int D>
// void FM structured mesh faces(const FM vector<B,FM u32>& dimensions, const std::vector<FM u64>* t, const FM submesh id* sid)
// {.....
//}
class FMstructured mesh iter impl : public FMiter impl
{

    struct FMstructured mesh phys to subsimplex
    
    template <int B, int D>
    
    int FM structured mesh phys to subsimplex(const FM phys<D>& p, FM context*, FM ptr<FM cell>*)
    
    const FM vector<B, FM u32>& dimensions,
    FM vector<B, FM u32>& s)
    
    FM structured mesh iter impl(const FMvector<B, FM u32>& b,
    const FM vector<B, FM u32>& s, const FM submesh id& submesh id, const FM vector<B, FM u32>& stride);

    protected:
    const FMvector<B, FM u32>& begin, end, stride;

    template <int B>
    
    class FMstructured mesh 0 cell iter impl:
    
    public:
    
    FMstructured mesh 0 cell iter impl(const FMtime<FM u32>& time,
    const FM submesh id& submesh id, const FM vector<B, FM u32>& stride);

    protected:
    const FMvector<B, FM u32>& begin, end, stride;

    template <int B>
    
    class FMstructured mesh B cell iter impl:
    
    public:
    
    FMstructured mesh B cell iter impl(const FMtime<FM u32>& time,
    const FM submesh id& submesh id, const FM vector<B, FM u32>& stride);

    protected:
    const FMvector<B, FM u32>& begin, end, stride;
else if (simplicial decomposition)
FM_iter impl ii;
#
for (i = 0; i < B; i++) {
#
if (cell type given) {
#
for (iter = ia.beginO; iter != ia.endO; ++iter) {
int n ignored = 0;
for (i = 0; i < B; i++)
//
FM vector<B,FM u32> strides;
for (i = 0; i < B; i++)
FM vector<B,FM u32> begin indices;
FM u32 cell
//bool
else if (cell dimension == B) {
}
}
}
}
}
}
return FM_iter(ii);
#
template <int B, int D>
FM_structured_mesh get_aux(const FM_structured_mesh<B,D>* sm, int pass, const FM_submesh_id* sid)
{
if (key == "name") {
std::vector<FM ptr<FM shared object> > values(B + i);
for (FM u32 i = 0; i < B; i++)
values[i] = new FM simple value<FM u32>(sm->get_card(i, 0, t, sid));
return new FM tuple value(values);
}
}
//
The generic FM_structured_mesh<B,D> declaration:
//
template <int B, int D>
class FM_structured_mesh : public FM mesh<B,m>
{
public:
const FM_vector<B,FM u32> dimensions;
protected:
}
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
}
//
The generic FM_structured_mesh<B,D> declaration:
//
void init(const FM_vector<B,FM u32>* = 0);
FMLStructuredMesh get_property_names(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);
return new FM tuple value(values);
public:

protected:

public:

private:

class FM structured mesh<3, m> : public FM mesh<3, m>
{
    template <int D>
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    virtual FM u64
    {
        FM structured mesh(FM properties cache* pc) :
        {
            void init(const FM vector<3, FM u32>& d)
            {
                FM structured mesh init(this, d);
            }
            FM vector<3, FM u32> dimensions;
            std::vector<FM u64> alignments[B + 1];
            FM u64 cube offsets[1 << B];
            get property names aux(std::set<std::string>& names,
            get aux(const std::string& key, FM u32 pass,
            iter begin() const
            iter begin(const FM iter attrs& ia) const
            phys to subsimplex(const FM phys<m>& p,
            phys to cell(const FM phys<m>& p, FM context* ctxt, FM ptr<FM cell>* c) const
            adjacencies(const FM cell* c) const
            faces(const FM cell* c, FM u32 k) const
            get structured behavior(const FM time<FM u32>* = 0,
            get base dimensions(const FM time<FM u32>* = 0,
            cell to enum(const FM cell* c) const
            enum to cell(FM u64 e, FM u32 d, FM u32 sd = 0,
            cell to subsimplex(const FM phys<m>& p, FM context* ct, FM ptr<FM cell>* c) const
            base to cell(const FM base<3, m>(pc),
            init(dimensions);
            public:
            virtual FM u64
            get card(FM u32 k, FM u32 sd : 0, const FM time<FM u32>* t : 0,
            get card(3, 0, t, sid); break;
            case 2:
                throw bad cell argument(this, "adjacencies", c);
            default:
                abort();
            return card;
            virtual FM ptr<FM cell>* faces(const FM cell* c, FM u32 d, FM u32 sd = 0,
            const FM time<FM u32>* = 0, const FM submesh_id* sid = 0) const
            return FM structured mesh faces(dimensions, alignments, c, k);
            virtual std::vector<FM ptr<FM cell>*> adjacencies(const FM cell* c) const
            return FM structured mesh adjacencies(dimensions, c, k);
            virtual std::vector<FM ptr<FM cell>*> faces(const FM cell* c, 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 faces(dimensions, alignments, c, k);
            virtual bool get_base_dimensions(const FM time<FM u32>* = 0,
            const FM submesh_id* = 0) const
            return return true;
            virtual FM u64 cell_to_index(const FM cell* c) const
            return FM structured mesh cell_to_index(alignments, dimensions, c, k);
        }
    }
}
virtual int base_to_cell(const FH mesh* m, FH u32 nind, const FH structured_B_cell<3>* sc) const
{
  return FH structured_mesh_base_to_cell(dimensions, b, sc);
}

virtual int phys_to_subsimplex(const FH mesh* m, const FH phys<3, p, FM_context* ctxt, const FH phys<3>* pm) const
{
  if (!pm)
    return 0;

  int rew = phys_to_base(p, ctxt, &b, &sc);
  rew += phys_to_base(p, ctxt, &s, &sm);
  return rew;
}

virtual int phys_to_subsimplex(const FH mesh* m, FH u32 nind, FH u64 ind[])
{
  return FH structured_mesh_phys_to_subsimplex(q, sm, ctxt, e);
}

virtual FM_iter_base begin() const
{
  return begin(_iter->attrs);
}

virtual int phys_to_cell(const FH phys<3>& p, FH context* ctxt, FH ptr<FH cell>* c) const
{
  FH ptr<FH structured_B_cell<3>*>(m);
  return FH structured_mesh_vertex_indices(const FH mesh*, n, FM_u32<2>* n_ind, FM_un4<2> ind[]);
}

virtual int phys_to_cell(const FH phys<3>& p, FH context* ctxt, const FH phys<3>* pm) const
{
  return FH structured_mesh_vertex_indices(const FH mesh*, const FH phys<3>* pm, FM_u32<2>* n_ind, FM_un4<2> ind[]);
}

const FH structured_mesh<3>* sm = reinterpret_cast<const FH structured_mesh<3>*>(m);
const FH structured_hexahedron_faces[dimension] aux;
for (int i = B - 1; i >= 0; i++)
  index = aux + dimensions[i];
index += _indices[i];

for (int i = B - 1; i >= 0; i++)
  index += indices[i] * sm->dimensions[i];
index += _indices[i];

return FH structured_mesh vertex_indices(const FH mesh*, n, FM_u32<2>* n_ind, FM_un4<2> ind[]);
template <>
void FM_structured_subsimplex<3> ::
structured_mesh_vertex_indices(const FM mesh * m, FMu32 * nind, 
FMu64 ind[]) const
{
    const FMu32* vi = FM_structured_hexahedron_subfaces(dimensions,subid);
    *nind = dimension + 1;
    const FMu64 index = indices[1] * m->dimensions[0] + indices[0];
    for (FMu32 i = 0; i < *nind; i++)
    ind[i] = index + m->cubeoffsets[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.
 */

#ifndef FM_SUBMESH_IDH
#define FM_SUBMESH_IDH

/*
 * NAME: FM submesh id.h
 * WRITTEN BY:
 * Patrick Moran
 * pmoran@nas.nasa.gov
 */

#include <iostream>
#include <stdexcept>
#include "FMtypes.h"

class FMsubmesh id
{
public:
    FMsubmesh id() : id(1) {
    FMsubmesh id(int i) : id(i) {
    FMsubmesh id(FMu32 i) : id(int(i)) {
    void set(FMu32 i) { id=int(i); 
    inline bool defined() const { return id ! 1; 
    friend std::ostream& operator<<(std::ostream& o, const FMsubmesh id& s)
    {
        if (s.defined())
            o << s.id;
        else
            o << "<submesh id undefined>";
        return o;
    }
    friend bool operator==(const FMsubmesh id& lhs, const FMsubmesh id& rhs)
    {
        return (lhs rhs);
    }
    friend bool operator<(const FMsubmesh id& lhs, const FMsubmesh id& rhs)
    {
        return lhs.id < rhs.id;
    }
    FMu32 index () const
    {
        if (!defined())
            throw std::logic_error("attempting to access undefined submesh id");
        return FMu32(id);
    }
private:
    int id;
};

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

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

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

FMu32 id() const
{
    if (defined())
        throw std::logic_error("attempting to access undefined submesh id");
    return FMu32(id);
}

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

template <typename T>
class FMtime {
public:
FMtime() : value(undefined_value) {
FMtime(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 setundefined() { value = undefined_value; }
inline bool defined() const { return value != undefined_value; }
friend bool operator==(const FMtime& lhs, const FMtime& rhs) 
{ return lhs.value == rhs.value; }
friend bool operator!=(const FMtime& lhs, const FMtime& rhs) 
{ return lhs.value != rhs.value; }
friend bool operator<(const FMtime& lhs, const FMtime& rhs) 
{ return lhs.value < rhs.value; }
T value;
static const T undefined_value;
};
#endif

template <typename T>
const T FMtime<T>::undefined_value = T(0);
#endif

class FMtimer {
public:
FMtimer() { reset(); }
void reset() { total = 0.0; }
void start() { gettimeofday(&starttv, (struct timezone *) 0); }
void stop() {
struct timeval stoptv;
gettimeofday (&stoptv, (struct timezone *) 0);
long dt = (stoptv.tv_sec - starttv.tv_sec) 
  * 1.0e6 + (stoptv.tv_usec - starttv.tv_usec) 
  * 1.0e3;
// round to milliseconds
long millisec = (long) (dt * 1.0e3 + 0.5);
total += millisec * 1.0e3;
}
friend bool operator< (const FMtimer& lhs, const FMtimer& rhs) 
{ return total < rhs.total; }
private:
};
#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

/ * 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

/ * 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

50
// Emacs mode * c++ *
#endif
/*
• NAME: FMtypes.h
• WRITTEN BY:
Patrick Moran pmoran@nas.nasa.gov
*/
#endif
#define FMCOORD
#define FM COORD
typedef float FM coord;
#endif
typedef unsigned short FMul6;
typedef unsigned FMu32;
typedef unsigned long long FMu64;
struct FMtruetype {
};
struct FMfalse_type {
};
template <int N, typename T> class FMvector;
template<typename T>
struct FM traits
{
    typedef T element_type;
    typedef FMfalse_type is_scalar;
};
template <>
struct FM traits<char>
{
    typedef char element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<unsigned char>
{
    typedef unsigned char element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<short>
{
    typedef short element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<unsigned short>
{
    typedef unsigned short element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<int>
{
    typedef int element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<unsigned int>
{
    typedef unsigned int element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<long>
{
    typedef long element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<unsigned long>
{
    typedef unsigned long element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<long long>
{
    typedef long long element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<unsigned long long>
{
    typedef unsigned long long element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<float>
{
    typedef float element_type;
    typedef FMtrue_type is_scalar;
};
template <>
struct FM traits<double>
{
    typedef double element_type;
    typedef FMtrue_type is_scalar;
};
#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
51
```
#ifndef FMVECTORH
#define FMVECTORH

/*
• NAME: FM vector.h
• WRITTEN BY:
Patrick Moran
*/

#include <iostream>
#include <utility>
#if defined( sgi) && !defined( GNUC )
#include <math.h>
#else
#include <cmath>
#endif
#include "FM types.h"

#include <list>
#include <utility>

pmoran@nas.nasa.gov

template <int N, typename T>
T FMdot (const FMvector<N, T>&, const FMvector<N, T>&) ;

template <typename T>
FM vector<3, T> FMcross (const FMvector<3, T>&,
const FMvector<3, T>&) ;

template <int N, typename T>
class FM vector
{
public :
FM vector() {
};
FM vector(const T dat[]) {
};
template <typename S>
explicit FM vector(const FM vector<N, S>& dat) {
};

T& operator[](int i) { return d[i]; } 
const T& operator[](int i) const { return d[i]; } 
typename FM traits<T>::element type* v() {
} 
const typename FM traits<T>::element type* v() const {
}

friend bool operator==(const FM vector<N, T>& lhs,
const FM vector<N,T>& rhs) {
}

FM vector<N,T>& operator+=(const FM vector<N,T>& v) {
} 
FM vector<N,T>& operator=(const FM vector<N,T>& v) {
} 
FM vector<N,T>& operator*=(typename FM traits<T>::element type s) {
} 
FM vector<N,T>& operator/=(typename FM traits<T>::element type s) {
}

friend FM vector<N,T> operator+(const FM vector<N,T>& u) {
} 
friend FM vector<N,T> operator+(const FM vector<N,T>& v) {
} 
friend FM vector<N,T> operator-(const FM vector<N,T>& u) {
} 
friend FM vector<N,T> operator-(const FM vector<N,T>& v) {
} 
friend FM vector<N,T> operator* (const FM vector<N,T>& u) {
} 
friend FM vector<N,T> operator* (const FM vector<N,T>& v) {
} 
friend bool operator==(const FM vector<N,T>& u,
const FM vector<N,T>& v) {
}

private:
T d[N];
}

template <typename T>
class FM vector<1, T>
{
public:
FM vector() {
} 
FM vector(const T dat[]) {
} 
template <typename S>
explicit FM vector(const FM vector<1,S>& dat) {
} 

T& operator[](int i) { return d[i]; } 
const T& operator[](int i) const { return d[i]; } 
typename FM traits<T>::element type* v() {
} 
const typename FM traits<T>::element type* v() const {
}

friend bool operator==(const FM vector<1,T>& lhs,
const FM vector<1,T>& rhs) {
} 

FM vector<1,T>& operator+=(const FM vector<1,T>& v) {
} 
FM vector<1,T>& operator=(const FM vector<1,T>& v) {
} 
FM vector<1,T>& operator*=(typename FM traits<T>::element type s) {
} 
FM vector<1,T>& operator/=(typename FM traits<T>::element type s) {
} 

friend FM vector<1,T> operator+(const FM vector<1,T>& u) {
} 
friend FM vector<1,T> operator+(const FM vector<1,T>& v) {
} 
friend FM vector<1,T> operator-(const FM vector<1,T>& u) {
} 
friend FM vector<1,T> operator-(const FM vector<1,T>& v) {
} 
friend bool operator==(const FM vector<1,T>& u,
const FM vector<1,T>& v) {
}
```

```
return
lhs.d[0] * rhs.d[0] +
lhs.d[l] * rhs.d[l];
}

// 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.

}