Field Model

The FM User Guide
Second Draft

Patrick J. Moran

NASA Ames Research Center, M/S T27A-2
Moffett Field, CA, 94035, USA
patrick.j.moran@nasa.gov

June 23, 2003

Contents

Chapter 1

Introduction

1.1 Overview

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 [20, 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.

For those in the visualization community, the large variety of mesh and field types offers the opportunities of new and interesting research topics. For example, with adaptive meshes one might want to couple various multi-resolution visualization techniques with the adaptive mesh data structures. For visualization system developers, the variety of mesh and field types are a challenge. There are a number of current development efforts, for example with adaptive meshes, 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 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.

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 following chapters we describe the design and implementation intended to make that goal a reality.

1.2 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 [8]. 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[20, 1]). Beginning with Haber et al [13], the fiber bundle model was adapted into a model that would support a general-purpose data-flow visualization system. OpenDX can handle "vertex-centered" and "cell-centered" fields. OpenDX does not support adaptive meshes, though more recent work by Treinish [30] describes a model that would accommodate such data.

Another field modeling effort was the Field Encapsulation Library (FEL) project, first presented at Visualization '96 [7]. 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 [24, 23]. FEL2 introduced fundamental design features that enabled the library to operate with far larger data sets, including a consistent demand-driven evaluation model [22] and the integration of demand-paging techniques [9]. FEL2, like the original version of FEL, assumed that all objects were in $\mathbf{R}^3$ physical space, and that all fields were "vertex-centered".

The Visualization Toolkit (vtk) [28], 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 [19] 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. The *FM* approach provides better insulation between data producers and consumers, which implies that new producers and consumers can be added in the future with less modifications to existing modules.

Another object-oriented data flow visualization system intended for large data visualization is SCIRun [5, 27]. 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 [16, 15] 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.

Chapter 2

Concepts

2.1 Cells and Meshes

Field Model objects are embedded in $\mathbf{R}^D$, also known as *physical space*. Objects in $\mathbf{R}^D$ are also said to have a *physical dimensionality* of D. Fields are based on meshes, which in turn are composed of cells. A *k-cell* is a subset of $\mathbf{R}^D$ that is homeomorphic (topologically equivalent) to a k-ball. Cells in *FM* are currently all linear objects. A 0-cell is a vertex, a 1-cell is an edge, 2-cells include triangles and quadrilaterals. Hexahedra, tetrahedra, pyramids and prisms are all examples of 3-cells. Every cell σ has a set of vertices. A *face* of σ can be specified by a non-empty subset of the vertices of σ [1]. For example, a hexahedron has vertex, edge, and quadrilateral faces. Every cell is also a face of itself. A *mesh* $\mathcal{M}$ is a finite collection of cells such that if $\sigma \in \mathcal{M}$, and τ is a face of σ, then $\tau \in \mathcal{M}$. Typically, cells in a mesh share common faces, so for example two tetrahedra can share triangle, edge, and vertex faces. If the cells with the highest dimensionality in mesh $\mathcal{M}$ are B-cells, then $\mathcal{M}$ is a B-mesh, and $\mathcal{M}$ has a *base dimensionality* of B. Meshes in *FM* are not allowed to have mixed dimensionality, in other words, every cell in a B-mesh $\mathcal{M}$ must be the face of some B-cell in $\mathcal{M}$. The base dimensionality of a mesh must be less than or equal to its physical dimensionality. Figure 2.1 illustrates example meshes that can be constructed in *FM*. Note that *FM* meshes can represent familiar objects such as regular meshes, curvilinear meshes, and tetrahedral unstructured meshes. *FM* can also represent objects most are less accustomed to thinking of as meshes, such as scattered vertices or a molecular skeleton. In most cases the shape of a B-mesh is a B-manifold, though for example the molecular mesh in Figure 2.1 would not qualify as a 1-manifold.

2.2 Fields

A *field* defines a function within a region of space. In *FM*, each field object has a set of values called *nodes* (which can be accessed on demand), a mesh, and a pairing between

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

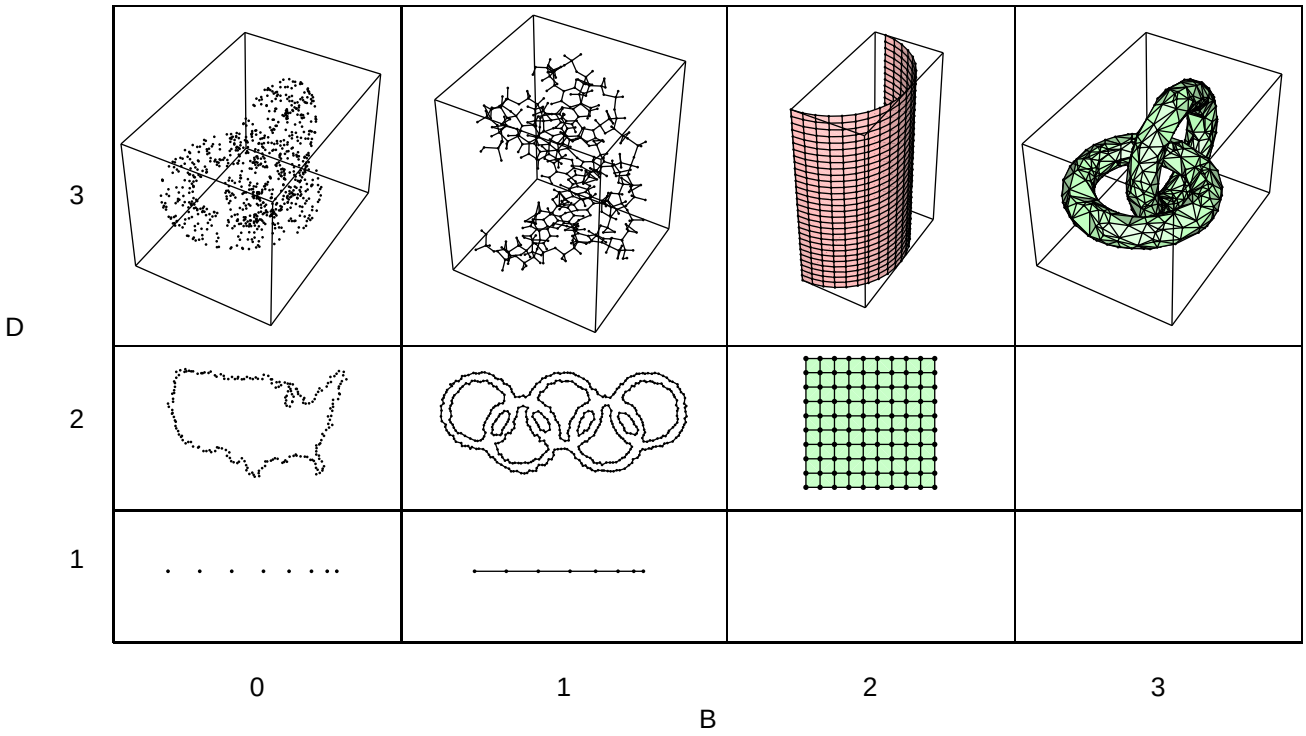

Figure 2.1: Example *FM* meshes organized in columns and rows by base dimensionality (B) and physical dimensionality (D), respectively. Note that the model is general enough to represent not only the input to visualization techniques, but also the output. For example, regular meshes in $\mathbf{R}^2$ ((B,D) = (2,2)) could serve as the underlying discretization for images, and surfaces in $\mathbf{R}^3$ naturally correspond to (2,3) meshes.

2-D v. 1-D time slice figure here

Figure 2.2: Caption here.

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.

2.3 Time

Given the general way *Field Model* is designed for handling data of various dimensionalities, one might wonder how time-varying data are handled. One approach would be to simply treat time as an extra dimension, reusing the existing multi-dimensional mechanisms. An alternative approach would be to keep time "special", in other words, a specifically identified parameter built-in to positional arguments. The *FM* design opts for the latter design, for several reasons:

- Most users are accustomed to an n-D plus time model. For example, our experience has been that most users do not prefer to think of a time-varying field

based on a hexahedral mesh as 4-D. *FM* uses some abstractions, such as that for cells, that require some users to think more n-dimensionally than they probably would otherwise, but in general we want abstractions that most users feel the most comfortable with.

- Time can be at a much lower resolution compared to spatial dimensions. In many simulation post-processing scenarios, the data available for analysis is already down-sampled in time, relative to the simulation. This implies that the user may wish to handle the time dimension differently for purposes of interpolation. This in turn does not necessarily mean that an explicit "time" parameter is required, but at least there should be a means to support interpolation that is not the same for every axis.

- Building higher-dimensional objects via Cartesian products can lead to awkward cell types. For example the Cartesian product of a tetrahedron with an interval would be a 4-cell that few would be eager to work with. Defining objects via Cartesian products does have a certain mathematical appeal, and several earlier data model articles proposed the approach [8, 13]. In *FM* we limit the use of the Cartesian product concept to defining structured mesh objects.

- For adaptive mesh applications, we would like to have a parameter to specify where in the series of mesh adaptations we were interested. Some applications may not think of that parameter being called "time", but the need for a parameter exists nevertheless.

- Implemenation challenges: we want abstractions that are general and powerful, but we also want an implementation that is relatively straight-forward to understand, and performance that is competitive. Clearly there are trade-offs, and we chose the route that we were more familiar with in terms of design and implementation.

Figure 2.2 illustrates how a time-varying field with 1 spatial dimension can be viewed either as 2-D data or as a 1-D slice

2.4 Simplicial Decomposition

Some algorithms require simplices, but not all meshes are composed solely of simplices. For example, some vector field topology algorithms require tetrahedra; if the mesh is hexahedral then we need to decompose the hexahedra into simplices. *FM* provides a means to request simplices, even when the mesh has other types of cells: simplicial decomposition. With simplicial decomposition turned on, *FM* mesh objects transparently decompose non-simplicial cells into simplicial cells. Internally, mesh objects typically support simplicial decomposition with relatively low overhead bookkeeping techniques, rather then constructing the full simplicial mesh. We revisit the simplicial decomposition topic in the structured and unstructured mesh chapters.

Chapter 3

Implementation Preamble

3.1 C++ and Templates

Field Model is implemented in C++. Familiarity with C++, templates in C++, and the C++ standard library classes (e.g., `std::vector<T>`) is essential. The *FM* implementation does not use some of the more exotic C++ template metaprogramming techniques (see for example [2]) such as expression templates [14, 31].

The *FM* implementation uses C++ namespaces to organize the code by modules. The primary namespace used by *Field Model* is the `FM` namespace. The definitions and examples shown in the following chapters are all assumed to be within `FM`. We revisit the topic of modules in Section 3.9 below.

3.2 Error Handling

Errors happen. In *FM* most errors are indicated by the integer return value provided by most object member functions. Success is indicated by the constant `OK`; various integer failure constants are defined in `FM_err.h`. *FM* objects may also generate exceptions in cases where there is not an opportunity to indicate an error via an integer return value. For example, if a problem is discovered during object construction, then that problem is indicated by an exception. *FM* exceptions are returned as either `std::runtime_error` or `std::logic_error` instances ([29], 14.10). Errors that correctly written code may encounter, such as problems opening a file, are reported as runtime exceptions. Errors that are due to incorrectly written code are reported as logic exceptions.

If one has a particular field class in mind when developing an application, then it is tempting to make "this call cannot fail" assumptions and skip checking return values. We encourage application developers to resist this temptation. One of the major benefits of the *FM* design is polymorphism. Polymorphism facilitates code reuse: in many cases we can transparently exchange one field instance for another that conforms to the same interface, without modifying the code written in terms of the interface. One

Abbrev.	C++ Type
i	int
u	unsigned
f	float
d	double

Table 3.1: *FM* abbreviations for scalar node types.

might know that a particular field class may never return a value other than `OK` for a specific call, but it is usually a mistake to assume all fields behave that way.

A second reason to be conscientious about return value checking is that not doing the checking can lead to potentially insidious bugs. Since most member functions work by writing their results into a location passed into the call, one always has something in result location, whether or not the call succeeds. In some cases it may be obvious that the result location contains junk, but at other times the contents may seem plausible. For example, the result may contain a value from a previous, successful call. Checking return values is the only reliable way to guard against this type of problem.

3.3 Coordinate Type

The coordinate type in *FM* is provided by the `Coord typedef`. The default for `Coord` is `float`. The implementation should also work with a `double` coordinate `typedef`. The type declaration provides a means of comparing how the coordinate type choice effects numerical accuracy and performance, albiet after recompiling the whole enchilada after the type is changed. Use of a single `typedef` means that one cannot easily mix objects with different coordinate types in the same application.

3.4 Vectors and Matrices

Small, fixed-length vectors appear throughout the *FM* design. In *FM* such vectors are represented by `Vector<N,T>` objects, where `N` specifies the vector length, and `T` specifies the element type. Most of the standard infix operators that one would expect for vectors are implemented for `Vector<N,T>` objects. One can also reference vector elements using square brackets notation. For the full `Vector<N,T>` declaration, see `FM_Vector.h`. The *FM* implementation provides `typedef` statements for some of the most frequently used `Vector<N,T>` instantiations. The naming convention concatenates `Vector`, the length, and a scalar type abbreviation from Table 3.1. For example, the `typedef` for a vector of 3 floats is `Vector3f`.

Matrices in *FM* are constructed as vectors of vectors. *FM* provides basic operations for matrices, such as multiplication, determinants and inversion. See `FM_Matrix.h`. Template notation for vectors of vectors gets pretty unwieldy, so *FM* provides `typedef` statements for the most commonly used instantiations. The convention is similar to that for `Vector` types. `Matrix`MNT stands for an $M \times N$ matrix, with an element type

abbreviation from Table 3.1. For example, `Matrix33d` is shorthand for a 3×3 matrix of doubles.

3.5 Multi-Threading

The *Field Model* implementation is intended to support multi-threaded applications. Introducing multi-threading immediately brings in other practical implementation issues. Some synchronization primitive libraries are not mutually compatible, thus committing to one set of primitives may make use with other libraries problematic. Furthermore, some frameworks employ relatively elaborate thread scheduling schemes that require buy-in from all components in the framework. We in *FM*-land take a minimal stand on this issue. *FM* has a single mutual exclusion primitive, based on `pthread_mutex_t`. See `FM_Mutex.h`. We defer on trying to provide a more full-featured set of thread synchronization abstractions for now.

3.6 Shared Objects

FM provides a relatively standard "smart pointer" reference-counted object mechanism (see for example, Meyers [21]) to facilitate sharing objects. *FM* implements a templated pointer type `Ptr<T>` and a class from which all shared objects inherit: `Object`. The reference counting mechanism is thread-safe. The most widely used shared object classes in *FM* are cells, meshes and fields. Meshes and fields are for the most part read-only, scanning through the mesh and field member functions one can see that they are almost all declared `const`. Sharing a mesh or field object among many threads is relatively straight-forward.

Working with `Ptr<Cell>` instances requires a little more awareness. Cell objects have many methods that explicitly change the state of the object (i.e., methods that are not declared `const`). Typically, an application that changes the that state of a cell object does not intend that object to be shared. Cells provide a `copy` method to create copies when needed; it is the application developer's responsibility to be aware of when to make a copy.

An earlier version of the *FM* implementation did not provide non-`const` member functions for cells or other shared objects. To change the state of a cell one had to construct a new cell with the desired state. This constraint protected the programmer from some potentially subtle bugs, but it came at a significant performance cost. For some common cell uses, such as iteration, the costs add up quickly. The *FM* implementation now relaxes the "shared objects must be immutable" requirement, but we add a few cautionary reminders later in this document, where appropriate.

3.7 Memory Management

FM objects allocate and deallocate memory using the standard C++ `new` and `delete` operators. The *FM* implementation follows the convention that when an object explicitly allocates memory, that object is responsible for eventually deallocating that mem-

ory. The one significant exception to this rule is with object construction: when an object such as a curvilinear mesh (Chapter 6) or core field (Chapter **??**) is constructed, the data array buffer passed in at construction time become the responsibility of the constructed object to eventually deallocate.

Occasionally one may want to suppress the automatic array buffer deletion. For example, if the data used to construct a core field are in a buffer shared with other application threads (e.g., an analysis application running concurrently with a simulation application), then it is unlikely that the user would want the *FM* object to deallocate it. Whether or not the buffer is shared, *FM* assumes that the buffer was allocated using C++ `operator new [] ()`, therefore the deallocation is done with C++ `operator delete [] ()`. Thus even if a buffer is not shared, one must suppress deletion by *FM* if the allocation was done another way; in general it is a bad idea to not use matched allocation and deallocation calls. *FM* provides delete supression via setting the `delete_suppression` property:

```
Ptr<Object> t = new Simple_Value<bool>(true);
object_with_buffer_responsibility->set("delete_suppression", t);
```

The *FM* properties mechanism is described later in Section 5.1 and Section 5.9.

3.8 Context

Field objects tend to be large, and for the most part, immutable. They are natural candidates for sharing when writing multi-threaded applications. One potential problem with sharing is that different threads may want the same field to behave in different ways. For example, different threads may require different interpolation modes. Or, one thread may require that simplicial decomposition (Section 2.4) be turned on, while another may require that it be turned off. *FM* provides `Context` objects to enable customization of behavior. Each thread in a multi-threaded application should have its own `Context`, and each can set its parameters as needed. Field methods, such as `at_cell`, take a pointer to a `Context` instance as an argument. Thus the same call made by different threads can behave differently, depending on the `Context` settings.

Threads should not share `Context` instances, even if they intend to use the same settings. Some *FM* mesh classes use `Context` objects to cache information, such as the last cell searched in point location. The caching will not work properly if multiple threads share the same `Context` instance.

Users familiar with FEL [24] will recognize that the `Context` design is one of the differences between FEL and *FM*. The lack of something equivalent to a `Context` object in FEL was a problem that lead to some fairly awkward solutions, none very satisfactory.

3.9 Module Organization

There are a multitude of file formats in use by scientists for storing data. Some are fairly widely recognized standards, such as PLOT3D [32], FITS [11], or HDF [12], others may be local to a particular research laboratory. *FM* does not define its own file

Module	Description
FITS	reads FITS files [11], metadata read into attributes
FM	the central *Field Model* module
HDF4	rudamentary support for HDF 4.x [12]
HDFEOS4	reads HDFEOS files using EOS library, based on HDF 4.x [26]
PLOT3D	extensive support for PLOT3D data [32]
SILO	the SILO format is native to LLNL, a minimal skeleton currently
VISUAL3	reads VISUAL3 unstructured meshes

Table 3.2: Current *FM* modules.

format, it defines interfaces and classes. The central *FM* interfaces are intended to be file-format neutral. While *FM* strives for file-format independence, from a practical standpoint there needs to be a relatively straight-forward means for scientists to be able to import and export data between their preferred data format and *FM* objects. Data objects are much more interesting to a scientist if they contain his or her own data. *Field Model* is organized as modules. The central FM module contains interfaces and classes common to various data standards. For data from specific file formats, there are modules corresponding to those formats. Table 3.2 lists the current set of *FM* modules. We describe the modules beyond *FM* in later chapters.

Chapter 4

Positional Classes

FM has three positional types: physical, base, and cell. All three types have a `Time` data member. `Base<B>` objects also have a `Sub` data member for specifying a subblock in a multi-block object. We describe the three positional types and the `Sub` and `Time` modifier types next.

4.1 `Phys<D,C>`

`Phys<D,C>` class instances are used to represent points in D-dimensional physical space. The second template parameter `C` specifies the coordinate type, it defaults to `Coord`. Currently *FM* only uses physical positions with the default coordinate type, thus physical position declarations appear with a single parameter. An abbreviated version of the class declaration looks like:

```
template <int D, typename C = Coord>
class Phys : public Vector<D,C>
{
public:
  Phys() {}
  Phys(const Vector<D,C>&);
  Phys(const Vector<D,C>&, const Time&);
  Phys(const Time&);
  const Time& get_time() const;
  void set_time(const Time&);
private:
  Time time;
};

template <int D, typename C>
std::ostream& operator<<(std::ostream&, const Phys<D,C>&);
```

See the full declaration in `FM_Phys.h`. `Phys<D,C>` is derived from `Vector<D,C>`. Like vector instances, individual coordinates can be accessed and set via the square brackets operator. Also, like a vector instance, a physical position instance can be

used as an argument to libraries expecting a pointer to a C-style array, For example, a physical position `p` with dimensionality 3 and a coordinate type of `float` could be used to provide coordinates to OpenGL via `glVertex3fv(p)`. Physical positions include a time data member.

4.2 Base<B,C>

`Base<B,C>` class instances are used to represent points in B-dimensional base space. As with the physical position class, the second template parameter defaults to `Coord`; currently *FM* leaves the second parameter at its default value, thus base position declarations typically appear with a single parameter. An abbreviated version of the class declaration looks like:

```
template <int B, typename C = Coord>
class Base : public Vector<B,C>
{
public:
  Base();
  Base(const Vector<B,C>&);
  Base(const Vector<B,C>&, const Sub&, const Time&);
  const Sub& get_sub() const;
  void set_sub(const Sub&);
  const Time& get_time() const;
  void set_time(const Time&);
private:
  Sub sub;
  Time time;
};

template <int B, typename C>
std::ostream& operator<<(std::ostream&, const Base<B,C>&);
```

See the full declaration in `FM_Base.h`. Unlike physical position objects, `Base<B>` include a `Sub` data member which is used to specify a subblock in a multi-block mesh. Base positions also include a time data member.

4.3 Cell

The `Cell` class is the parent class to a variety of cell subclasses in *FM*. `Cell` is an abstract class that declares interface and data members common to all concrete cell classes, i.e., the classes that one can directly instantiate. We distinguish between classes and types here: there are more cell classes than there are cell types. For example, there are different vertex cell classes for structured and unstructured meshes. We introduce the specific cell classes in the following chapters along with specific mesh types. Table 4.1 summarizes the current set of cell types in *FM*, along with their dimensionality and face counts.

An abbreviated version of the `Cell` declaration looks like:

Prefix	Dim.	Type	0-faces	1-faces	2-faces	3-faces
V	0	VERTEX_CELL	1	0	0	0
E	1	EDGE_CELL	2	1	0	0
F	2	TRIANGLE_CELL	3	3	1	0
Q	2	QUADRILATERAL_CELL	4	4	1	0
T	3	TETRAHEDRON_CELL	4	6	4	1
P	3	PYRAMID_CELL	5	8	5	1
W	3	PRISM_CELL	6	9	5	1
H	3	HEXAHEDRON_CELL	8	12	6	1

Table 4.1: The *FM* cell types. "Prefix" refers to the prefix used with cells are written to an `std::ostream`. The prefixes are chosen to be unique. "F" stands for "Face" or "Facet", "W" stands for "Wedge". The "Dim." column contains the values that would be returned by `get_dimension()`. The type constants, such as `VERTEX_CELL`, would be returned by `get_type()`. The final 4 columns contain the return values for `get_n_faces(`k`)` for k equal to 0 through 3. Thus for example a prism has 6 0-faces (vertices), 9 1-faces (edges), 5 2-faces (triangles and quadrilaterals) and 1 3-face (itself).

```
class Cell : public Object
{
public:
  const Sub& get_sub() const;
  const Time& get_time() const;
  virtual void set_sub(const Sub&);
  virtual void set_time(const Time&);

  virtual unsigned get_dimension() const = 0;
  virtual unsigned get_n_faces(unsigned) const = 0;
  virtual int get_type() const = 0;
  virtual bool is_subsimplex() const;
  virtual Cell* copy() const = 0;

  friend bool operator==(const Cell&, const Cell&);

protected:
  Cell();
  Cell(const Sub&, const Time&);

  Sub sub;
  Time time;
};

bool operator!=(const Cell&, const Cell&);
```

The full declaration for `Cell` and the subclasses of `Cell` can be found in `FM_Cell.[hC]`.

4.4 Sub

Currently *FM* provides support for multi-block objects via the `Multi_Mesh<B,D>` and `Multi_Field<B,D,T>` classes (see Chapter **??**). The current classes provide a single level of hierarchy, thus the minimum required to specify a subblock is a single integer value. Compared to using a "bare" integer, the `Sub` class provides safeguards against using a subblock value that is not initialized. In the longer run, the `Sub` class is intended to be able to grow to support specifying subblocks in more general hierarchical objects, such as adaptive meshes. An abbreviated version of the `Sub` declaration looks like:

```
class Sub {
public:
  Sub();
  Sub(const unsigned);
  bool defined() const;
  operator unsigned() const;

  friend bool operator==(const Sub&, const Sub&);

private:
  static const unsigned UNDEFINED;
  unsigned sub;
};

bool operator!=(const Sub&, const Sub&);
std::ostream& operator<<(std::ostream&, const Sub&);
```

The full declaration for `Sub` can be found in `FM_Sub.[hC]`.

4.5 Time

FM uses three time representations: physical, base and step. (For an introduction to time in *FM*, see Section 2.3.) The step representation is essentially an integer version of base representation. The *FM* design allows one to use whichever representation works best for their application. Currently, most time-varying objects internally consist of a series of instances in time. See `Time_Series_Mesh<B,D>` and `Time_Series_Field<B,D,T>`. Setting the time member of a positional object to a physical value provides the most generality: one can query a time-series object without concern to where in time the individual time steps are located. Using physical time implies a bit more cost per access: time-series objects must locate where a given physical time falls in the series and possibly do temporal interpolation. The small amount of extra computational cost can add up if the positional object is used in many calls. We return to this issue in the time-series chapter.

The base time representation is in some respects an intermediate to physical and step representations. Like spatial base coordinates, a base time coordinate provides a means for the user to specify a position with respect to the discretization. For example, a base time of 0.5 would be half way between the first and second steps in a time series.

(Base coordinates are C-style, starting at 0). As with physical coordinates, access using base coordinates may require a time-series object to do temporal interpolation.

The third time representation is step. Like base coordinates, the step representation is discretization aware. Unlike base coordinates, step coordinates are restricted to integer values. The step time representation provides a means for the user to specify a particular instance in a time series with the assurance that no temporal interpolation will occur. Using step time in a position argument is the cheapest per access.

Time representation in *FM* is implemented by the class `Time` and three classes derived from `Time`: `Phys_Time`, `Base_Time` and `Step_Time`. An abbreviated version of the class declarations looks like:

```
class Time {
public:
  Time();
  Time(const Time&);
  bool defined() const;
  friend bool operator==(const Time&, const Time&);
protected:
  enum Representation {UNDEFINED, PHYS, BASE, STEP};
  Representation representation;
  union {
    Coord f;
    unsigned u;
  } u;
  Time(const Representation&, Coord);
  Time(const Representation&, unsigned);
};

bool operator!=(const Time&, const Time&);
std::ostream& operator<<(std::ostream&, const Time&);

class Phys_Time : public Time
{
public:
  Phys_Time(Coord t);
};

class Base_Time : public Time
{
public:
  Base_Time(Coord t);
};

class Step_Time : public Time
{
public:
  Step_Time(unsigned t);
};
```

The full declaration for the time classes can be found in `FM_Time.[hC]`. Time objects

provide an equality test (`operator==`). Two time objects are equal if and only if they have both the same representation and same the value. Mixing time representations in an application should be done with caution if one requires equality tests between positional arguments. For example:

```
Vector3u indices(7, 11, 13);
Ptr<Cell> u = new Structured_0_Cell<3>(indices);  // vertex
Ptr<Cell> v = u->copy();
std::cout << (*u == *v ? "true" : "false") << std::endl;

u->set_time(Step_Time(1));
v->set_time(Base_Time(1.0));
std::cout << (*u == *v ? "true" : "false") << std::endl;
```

The first output statement returns true, the second false. Conceivably the *FM* implementation could go to greater lengths to ascertain whether a time in step representation is equivalent to a time in base representation, but it does not. In general, the equality test code cannot compare times in two different representations because the means to convert between representations is not available at the point of the equality test.

Here are a few more typical usage examples involving time:

```
Phys<3> p;
Base<3> b;
p.set_time(Step_Time(2));
b.set_time(p.get_time());

bool equal = p.get_time() == b.get_time();
std::cout << (equal ? "true" : "false") << std::endl;
```

The final statement prints `true`.

Chapter 5

The Mesh and Field Interface

A key interface in *Field Model*, as one would expect, is that for fields. The interface is templated by three parameters: `B`, `D` and `T`. `B` and `D` are integer parameters specifying base and physical dimensionality, respectively. `T` specifies the field node type, e.g., `float`. The interface is declared by the class `Field<B,D,T>`. The subset of the interface that is not dependent on `B`, `D` and `T` is declared by the class `Field_` (parent class to `Field<B,D,T>`). The `Field_` type provides a convenient means to handle fields in a generic manner, for example, when one has a collection of fields with mixed dimensionalities and node types.

Every field has a mesh; the mesh encapsulates both geometric and topological information. The mesh interface is the same as for fields, and in fact every mesh *is a* field. As with other fields, the `B` and `D` parameters specify base and physical dimensionality. The node type for meshes is the type for coordinate vectors: `Vector<D,Coord>`. The "every field has a mesh" principle still holds true: a mesh is its own mesh.

The field interface methods are covered in the following sections:

Section 5.1 Properties (`get_base_dimensionality`, etc.)

Section 5.2 Cardinality (`card`)

Section 5.3 Canonical enumeration (`enum_to_cell`, `cell_to_enum`)

Section 5.4 Cell incidence relationships (`closure`, `star`, `faces`)

Section 5.5 Cell neighbor relationships (`neighbors`)

Section 5.6 Field values (& coordinates) (`at_base`, `at_cell`, `at_phys`, `at_vert`)

Section 5.7 Converting between positional representations (`phys_to_cell`, etc.)

Section 5.8 Iterators (`begin`, `split_begin`, `end`)

Section 5.9 General properties interface (`get`, `set`)

Section 5.10 Miscellaneous.

Specific mesh and field classes are covered in the following chapters.

Property	C++ Return Type	get_* Arguments
base_dimensionality	unsigned	
blanking_behavior	bool	const Sub* =0, const Time* =0
bounding_box	std::pair<VDC,VDC>	const Sub* =0, const Time* =0
dimensions	Vector<B,unsigned>	const Sub* =0, const Time* =0
mesh	Mesh<B,D>*	const Sub* =0, const Time* =0
min_max	std::pair<T,T>	const Sub* =0, const Time* =0
n_subs	unsigned	const Sub* =0, const Time* =0
node_association_index	unsigned	
node_type	std::string	
phys_dimensionality	unsigned	
property_names	std::set<std::string>	const Sub* =0, const Time* =0
structured_behavior	bool	const Sub* =0, const Time* =0
time_varying_behavior	bool	const Sub* =0

Table 5.1: The standard mesh and field properties that are accessible via get_* member functions. The template arguments B and D specify base and physical dimensionalities, respectively. The T template argument specifies field node type. The identifier VDC used in the bounding_box return type is shorthand for Vector<D,Coord>.

5.1 Properties

Properties are values associated with a field object such as the base dimensionality, a string describing the field node type, or the structured dimensions of an underlying mesh. For convenience and performance, *FM* provides dedicated access functions for the most frequently used properties. Field objects also have a general property mechanism, which we describe later in Section 5.9. Table 5.1 summarizes the properties available via dedicated functions. The access function name format is get_ followed by the property name, e.g., get_base_dimensionality. The get_ functions may also take optional arguments, listed in Table 5.1. We describe the properties next.

5.1.1 Behaviors

Users can query whether a mesh or field object has blanking, structured, or time_varying behavior using the corresponding get_* calls. Querying for a particular behavior is preferred to testing whether one has a particular implementation class (e.g., dynamic casting down to Structured_Mesh<B,D>*) because there are many ways to compose objects that have each of the recognized behaviors. "Blanking" indicates that the data contain a means of indicating nodes that do not have valid values. For example, remote-sensing data may specify a reserved value to use at nodes where data were not available. In *FM* requesting data based on blanked nodes produces the BLANKED_DATA return value. One can also query blanking behavior for a particular subblock in a multi-block object by defining the Sub argument. If the data are multi-block and Sub is not defined, then blanking_behavior is true if any of the subblocks have blanking behavior. For time-varying data, blanking_behavior is true if blanking_behavior is true at any point in time.

The `structured_behavior` property indicates whether methods exclusive to objects with structured behavior can succeed. Such methods include requesting the `dimensions` property and those involving an argument in base coordinates (i.e., a `Base<B>` argument). As with `blanking_behavior`, one can query a specific subblock in a multi-block object using the optional `Sub` argument. Unlike `blanking_behavior`, if the `Sub` argument is not defined and the object is multi-block, then `structured_behavior` is true if and only if *all* subblocks exhibit structured behavior. *FM* uses some interpolation and differential-operator techniques that only apply in structured-behavior cases. *FM* internally tests for `structured_behavior` as a prerequisite to using those techniques.

The `time_varying_behavior` property, as the name suggests, indicates whether the data vary with time. For multi-block data, `time_varying_behavior` is true if any of the subblocks are time-varying. There is no need for an optional `Time` argument for this property.

5.1.2 Dimensionality and Node Association Index

The methods `get_base_dimensionality` and `get_phys_dimensionality` provide a means to query the base and physical dimensionalities of a mesh or field object. Typically these properties are queried when starting with a `Field_` object, as a preable to casting and calling member functions where the arguments depend on the dimensionalities. The method `get_node_association_index` returns the node association, e.g., 0 for "vertex-centered" data. The dimensionalities and node association must be homogeneous across the subblocks of a multi-block object, thus there is no need to provide an optional `Sub` argument. These properties cannot change over time, thus there is no optional `Time` argument.

5.1.3 Node Type

The node type of a field can be requested in a `std::string` format. With multi-block objects the node type must be the same across all the subblocks. Furthermore, node type cannot change over time. Thus `get_node_type` has no optional arguments. The naming convention for scalar node types follows that used for elements in *FM* vector and matrix typedefs, see Section 3.4. For example, a field of C++ `float` values would have node type `"f"`. Node types corresponding to an *FM* `Vector<N,T>` are composed by concatenating `v`, N, and the standard *FM* name for type T, for instance `"v3f"`. Naming for vectors of non-scalar types is defined recursively, e.g., a 3×3 matrix of doubles would have the node type `"v3v3d"`.

5.1.4 Dimensions

The dimensions property applies to objects with structured behavior, e.g., a regular mesh. The dimension property is described in the structured meshes chapter (Chapter 6). For multi-block objects, `Sub` must be defined – there is no aggregate dimension property. Requesting the dimensions property on an object that does not have structured behavior will generate an exception.

5.1.5 Mesh

Every field has a mesh, and `get_mesh` provides a means to access that mesh. Calling `get_mesh` on a mesh causes the mesh to return itself. By defining the `Sub` argument one can access a submesh in a multi-block object. Through the `Time` argument one can access the mesh from an individual step in a time-series.

5.1.6 Bounding Box and Min-Max

The method `get_bounding_box` returns a pair of points in $\mathbf{R}^D$: the low and high corners of the mesh bounding box. The optional `Sub` argument can be used to restrict the query to a particular subblock in a multi-block object. The `Time` argument can be used to specify a particular instance in time. If the mesh varies with time, and no time is specified, then the bounding box over the whole time range will be returned.

The method `get_min_max` returns the minimum and maximum values for a field. If the field node type is non-scalar, then the first value in the returned pair contains the minimum value for each element, the second value in the returned pair contains the maximum value for each element. As with `get_bounding_box`, the optional `Sub` and `Time` arguments can be used to restrict the query to a specific subblock or instance in time. For mesh objects, `get_min_max` is the same as `get_bounding_box`.

5.1.7 Number of Subblocks

The `n_subs` property specifies the number of subblocks in a multi-block object. The property is 0 for objects that are not multi-block. The `Sub` and `Time` optional arguments become important with hierarchical and adaptive meshes. Currently *FM* provides meshes and fields with a single level of hierarchy, and the number of subblocks cannot change over time, thus the optional arguments do not yet come into play.

5.1.8 Property Names

The `property_names` property provides a means to query about what properties are defined by an object. At a minimum the property names listed in Table 5.1 will be returned. Various mesh and field subclasses are free to define extra properties. For example, PLOT3D [32] solution data includes a `reynolds_number` property for fields. The values associated with additional named properties can be requested via the general properties interface, described in Section 5.9 below. Through `property_names` one can do a form of object introspection.

5.2 Cardinality

Meshes are collections of cells. The cardinality, or quantity, of any particular cell type or cell dimensionality can be requested via the `card` member function:

```
unsigned long long card(int, const Sub* =0, const Time* =0,
                        unsigned=0, int=GROUP_UNDEFINED) const;
```

The first argument specifies a cell type or dimensionality. Typical cell types include `VERTEX_CELL` or `HEXAHEDRON_CELL`, the full list of cell type constants can be found in Table 4.1. If the first argument is an integer $k \geq 0$, then `card` returns the cardinality of k-cells. The distinction between cell types and cell dimensionalities becomes important when working with meshes that contain a mix of cell types for a given dimensionality, for example, an unstructured mesh where the 3-cells are a mix of prisms and hexahedra. With multi-block objects, `card` sums over the subblocks. Defining the `Sub` argument causes `card` to return cardinality for the given subblock only. The `card` interface is intended to support adaptive mesh uses. For adaptive meshes, the `Sub` argument would be used to specify a subtree in the adaptive mesh hierarchy. The `Time` argument would be used to specify the instance in time (where in the series of adaptations) that one is interested in. The second-to-last argument specifies simplicial decomposition, by default simplicial decomposition is off (0). The final optional argument is used to specify groups of cells identified beforehand by particular data formats. For example, PLOT3D [32] unstructured meshes identify sets of triangles that typically correspond to aircraft surfaces or symmetry planes.

5.3 Cells and Canonical Enumeration

For each type of cell in a mesh, *FM* provides a canonical enumeration, in other words, there is a unique integer associated with each cell of a given type. If the cardinality for a particular cell type is n, then the enumeration values are in the range $0..(n-1)$. The enumeration is useful for, among other things, bookkeeping. For example, an algorithm that keeps track of sets of cells can reduce memory usage by storing the enumeration value for each cell rather than the cell itself. To convert between one representation and the other, *FM* provides the methods:

```
int cell_to_enum(const Cell* c, unsigned long long* e,
                 const Sub* =0, const Time* =0) const;
int enum_to_cell(unsigned long long e, int cell_type, Ptr<Cell>* c,
                 const Sub* =0, const Time* =0) const;
```

For multi-block objects, the canonical enumeration is cumulative. If the `Sub` argument is defined, then the enumeration value will be with respect to the specified subblock only. The `Time` argument would be applicable in adaptive mesh cases, since the meshes and their enumerations would be changing over time.

The canonical enumeration chosen by various mesh and field objects is not arbitrary. Typically the numbering is chosen so that each value is equivalent to the index one would use to access data from a single array with the "usual" layout. We revisit the topic of canonical enumeration in later chapters.

5.4 Cell Incidence Relationships

Given a cell c, an application may require cells incident to c. Two cells c and d are *incident* if c is the face of d or vice versa. For example, for a given triangle c in a mesh, one may need the vertices of c, or perhaps the tetrahedra for which c is a

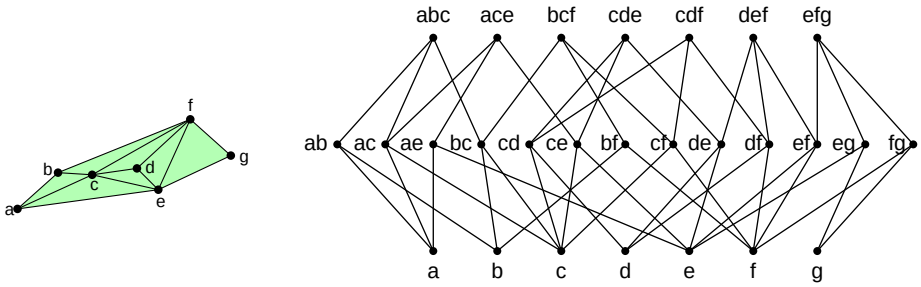

Figure 5.1: A small example mesh and its incidence graph.

face. The incidence relationships among cells can be described as a partial ordering (see for example Alexandroff [4]), and visualized as a graph. Figure 5.1 illustrates the incidence relationships for a small mesh in $\mathbf{R}^2$. The graph to the right contains a node for each cell in the mesh to the left. The nodes are organized into rows, each row containing cells of a particular dimension. The rows are ordered by ascending dimensionality: higher rows signify higher-dimension cells. A mesh containing 3-cells would have one extra row at the top. The example queries from above can be seen as starting at a particular node and following paths downward or upward. For example, to get the vertices of a triangle c, one could start at the node representing c and follow all the paths downwards. Likewise, in a tetrahedral mesh, one could start at a node representing a triangle c and follow the 1 or 2 paths upward, depending on the number of tetrahedra that have c as a face. *FM* meshes support queries based on cell incidence relationships via `closure`, `star` and `faces`:

```
int closure(const Cell* c, unsigned k, std::vector<Ptr<Cell> >*) const;
int star(const Cell* c, unsigned k, std::vector<Ptr<Cell> >*) const;
int faces(const Cell* c, unsigned k, std::vector<Ptr<Cell> >*) const;
```

The closure and star function definitions follow those in algebraic topology texts (see for example [3, 4, 25]) except that the methods return cells of a single dimensionality k. For a j-cell c, *closure* returns the k-faces of c, where $k \leq j$. For example, one can use `closure` to obtain the vertex faces of any cell. For a j-cell c, *star* returns the k-cells that c is the face of ($k \geq j$). For example, the `star` of a quadrilateral c in a hexahedral mesh can return the 1 or 2 hexahedra that c is the face of. The `faces` method combines the functionality of `closure` and `star`: j-`faces` of a k-cell is equivalent to `closure` if $j < k$ and `star` if $j > k$. In the the case where $j = k$, the `faces` result would contain c; every cell is trivially a face of itself.

Note that the concepts of incidence relationships, closure and star are not specific to a particular type of mesh; algorithms written in terms of `closure` and `star` have the potential of working with many types of meshes.

5.5 Neighbor Relationship

A concept related to the incidence relationships between cells is neighbor relationship. The method `neighbors` supports requests for the cells neighboring a given cell:

```
int neighbors(const Cell*, std::vector<Ptr<Cell> >*) const;
```

The typical usage of the `neighbors` with a B-mesh is with a B-cell argument. For example, given a hexahedron c from a hexahedral mesh, `neighbors` would return the hexahedra which share a quadrilateral face with c. Likewise, given a tetrahedron c from a tetrahedral mesh `neighbors` will return the tetrahedra which share a triangle face with c. The `neighbors` method is handy for algorithms that work breadth-first, starting from a seed cell. For example, one could construct an isosurface incrementally, processing cells outward from an initial 3-cell.

We use a formal definition for `neighbors` following that given for "adjacent cells" in Dobkin and Laszlo [10]. Returning to the graph in Figure 5.1, imagine that each vertex is the face of a special (-1)-cell, i.e. that there is an extra row beneath the vertex (0-cell) row with one node, and arcs from each vertex to the (-1)-cell node. Furthermore, if the cells in the top row are k-cells, then imagine an extra row above the k-cells with a single $(k+1)$-cell that every k-cell is the face of. Given this augmented incidence relationship graph, one can define the adjacent cells of a cell c via the star and closure operations described above. Let S_{ud} be the set of cells produced by going up one dimension and then down one dimension, starting with c. Let S_{du} be the set of cells produced by going down one dimension and then up one dimension, starting with c. The *neighbor* cells to a cell c are the cells in $(S_{ud} \cap S_{du}) - c$. For example, for the mesh in Figure 5.1, the neighbors of vertex c would be the vertices a, b, d, e and f. In the molecular skeleton in Figure 2.1, the neighbors of a vertex a would be the vertices that share an edge with a, i.e., the neighbors of atom a would be the atoms that share a bond with a.

5.6 Accessing Field Values and Coordinates

The field interface provides field value access methods for each of the three positional types: `Base<B>`, `Cell`, and `Phys<D>`. The interface also provides an additional method dedicated to a specific cell type: vertex. The signatures for the four methods are:

```
int at_base(const Base<B>&, Context*, T*) const;
int at_cell(const Cell*, Context*, T*) const;
int at_phys(const Phys<D>&, Context*, T*) const;
int at_vert(const Cell*, Context*, T*) const;
```

We describe each method in detail next.

5.6.1 `at_base`

The `at_base` method provides access to the value of a field at a specific `Base<B>` position. The field must have the `structured_behavior` property. Internally, `at_base`

entails point location and interpolation. Calling `at_base` on a mesh provides a means to convert spatial coordinates from base to physical.

5.6.2 at_cell

The `at_cell` method is a "workhorse" method that many other methods operate in terms of. The number of values written into the buffer passed in as the final argument to `at_cell` depends on the type of the cell argument and the node association index of the field. For a field where the field values are in a single buffer (`data` in the example below), such as a `Core_Field_T_Layout<B,D,T>` instance (see Chapter **??**), `at_cell` is defined as:

```
at_cell(const Cell* c, Context*, T* vals) const
{
  std::vector<Ptr<Cell> > faces;
  int res = mesh->faces(c, node_association_index, &faces);
  if (res != OK)
    return res;
  for (size_t i = 0; i < faces.size(); i++) {
    unsigned long long index;
    res = mesh->cell_to_enum(faces[i], &index);
    if (res != OK)
      return res;
    *vals++ = data[index];
  }
  return OK;
}
```

The `faces` method is used to obtain the cells that are paired 1-to-1 with the field nodes, then `cell_to_enum` is used to obtain the data array index for each one. Thus for example if the argument were a hexahedron, and the node association index was 0, then `faces` would return the vertex faces, and 8 field values would be written into the `vals` buffer. If the first argument were a vertex in a "cell-centered" (i.e., node association index 3) field based on a hexahedral mesh, then 1 to 8 values would be written in the `vals` buffer, depending on the number cells produced by the `star` of the vertex.

The definition above is general but rather expensive to evaluate. Many classes implement `at_cell` using algorithms that exploit optimization opportunities, see for example `Classic_Core_Field_T_Layout<B,D,T>` (Chapter **??**).

In most cases the number of field values to be written into the buffer provided as a final argument to `at_cell` is known *a priori*. In cases where the number of values to be written is not known, there is the method `n_at_cell`. See Section 5.10.

5.6.3 at_phys

The `at_phys` method is used to obtain a field value at a specific position in physical space. Internally, `at_phys` entails point location and interpolation. The `at_phys` method also has a use with meshes: by the return code one can determine whether a given physical point is within the field domain or not.

5.6.4 at_vert

The at_vert method is like at_cell, except that the first argument is assumed to be a vertex class instance and the field is assumed to have a node association index of 0. Any call to at_vert could be replaced with at_cell; at_vert is provided because accessing "vertex centered" data one vertex at a time is a common usage scenario, and we wanted to provide optimized performance for that scenario. Observant readers will notice that the first argument to at_vert is declared as const Cell* rather than a vertex class. This is because there is more than one class for representing vertices: Structured_0_Cell<B> for structured meshes, Unstructured_Vertex for unstructured meshes. Structured_K_Cell<B> is also an option for structured meshes. Internally at_vert uses C++ reinterpret_cast to cast the cell argument to an appropriate vertex cell class. The user is responsible for providing the right cell type argument to at_vert, the use of reinterpret_cast means that the C++ compiler will not protect the user from type mistakes here. Either a Structured_0_Cell<B> or Structured_K_Cell<B> instance will work with structured meshes. An *FM* iterator is guaranteed to provide an appropriate type if it is initialized to iterate over vertices (by the same field as that used for the at_vert calls).

5.7 Converting Between Positional Representations

The field interface provides a means of converting from one of the three positional representations to another:

```
int base_to_cell(const Base<B>&,
                 Ptr<Structured_B_Cell<B> >*) const;
int phys_to_cell(const Phys<D>&, Context*, Ptr<Cell>*) const;
int phys_to_base(const Phys<D>&, Context*, Base<B>*,
                 Ptr<Structured_B_Cell<B> >* =0) const;
```

Fields delegate these methods to their underlying mesh. We discuss each method, and the conversion combinations that the interface does not provide, below.

5.7.1 base_to_cell

The call base_to_cell produces the B-cube from a B-mesh that contains the given base position.

5.7.2 phys_to_cell

The method phys_to_cell essentially does point location, producing a cell that contains a given point p in physical coordinates. Note we say *a* cell rather than *the* cell because it possible for cells to overlap, especially with multi-block objects. Point location is not defined for fields where the base dimensionality is not equal to the physical dimensionality. For example, point location is not defined for a surface in $\mathbf{R}^3$.

5.7.3 phys_to_base

The method phys_to_base converts a position from one space to the other. The optional final argument provides an opportunity to obtain a cell containing the given physical point. This cell is the same as one would obtain using base_to_cell. As with phys_to_cell, phys_to_base is only defined in cases where the physical and base dimensionalities are equal.

5.7.4 The Other Combinations

One can imagine six possible combinations, three beyond the three listed above. The three additional methods would be "base_to_phys", "cell_to_phys" and "cell_to_base". The effect of "base_to_phys" can be achieved by calling at_base on the underlying mesh, i.e.:

```
// equivalent to "base_to_phys"
res = mesh->at_base(b, ctxt, &p);
p.set_time(b.get_time());
```

The methods "cell_to_phys" and "cell_to_base" are ambiguous since in general cells do not correspond to a single point in physical or base space.

5.8 Iterators

Fields are based on meshes, and meshes are collections of cells. *FM* provides iterators to ease the development of algorithms that operate over sets of mesh cells. The interface is intended by be familiar to those already familiar with C++ standard library iterators, but there are some differences due to the nature of how one might want to iterate over mesh and field objects:

- Meshes are collections of cells, and those collections include a variety of cell types; C++ standard library collections contain only one type. Users typically want to iterate over only one cell type at a time.
- Users often want to iterate over a subset of cells; a means to efficiently describe a subset at iterator initialization time is needed.
- C++ iterators are not polymorphic, but we want polymorphism. We want to be able to iterate over a mesh and use a single iterator to access coordinates and field values. In some cases we may have multiple fields, possibly with different node types, based on the same mesh. We may want to use the same iterator to access data from each.

FM supports iteration with the methods begin, split_begin, and end. The declarations for the three methods look like:

```
Iter begin(const Iter_Attrs* =0) const;
std::vector<Iter> split_begin(unsigned n, const Iter_Attrs* =0) const;
Iter end() const;
```

`Iter_Attrs` is essentially a `std::vector<Iter_Attr>`, and one can use standard vector methods to add `Iter_Attr` instances to the vector. The method `split_begin` is intended for multi-threaded applications. Given a first argument n, `split_begin` returns n iterators, each for roughly the same number of cells. How the split is done depends on the underlying mesh type. The `split_begin` method provides a means to load balance work over multiple threads. Finally, the `end` method is used in the same way as with standard library collections.

The basic use of *FM* iterators looks like:

```
for (Iter iter = field->begin(); iter != field->end(); ++iter) {
  int res = field->at_cell(*iter, &ctxt, &val);
  ...
}
```

By default *FM* iterators iterate over all the vertices in a mesh. Dereferencing the iterator produces an argument of type `const Cell*` for use with `at_cell` or `at_vert`, where appropriate.

A slightly more advanced example of iterator usage would be:

```
Iter_Attrs iter_attrs;
iter_attrs.push_back(Iter_Attr(CELL_TYPE, HEXAHEDRON_CELL));
for (Iter iter = field->begin(&iter_attrs); !iter.done(); ++iter) {
  int res = field->get_mesh()->at_cell(*iter, &ctxt, xyzs);
  ...
  res = field->at_cell(*iter, &ctxt, vals);
  ...
}
```

In this case we use an `Iter_Attrs` instance to specify that we want to iterate over hexahedra. Note too that we test whether we are done using the expression `!iter.done()` rather than comparing the iter with `field->end()`. Using the `done` method is more efficient. There are many more attributes that we can specify via `Iter_Attr` instances; the choices are dependent upon the mesh type. We revisit iterator initialization in later chapters. Within the loop body, note that we can use the same iterator as an argument to both the underlying mesh and to the field.

Another usage scenario that is fairly common is iterating over cells to identify those that meet some criterion. For example, when using simplicial decomposition with a hexahedral mesh, we will get tetrahedra with zero volume if there are hexahedra with collapsed edges. Here is an excerpt where we search for those cells:

```
Iter_Attrs iter_attrs;
iter_attrs.push_back(Iter_Attr(SIMPLICIAL_DECOMPOSITION, 1));
Context ctxt;
double volume;
std::vector<Ptr<Cell> > zero_volume_cells;
for (Iter iter = mesh->begin(&iter_attrs); !iter.done(); ++iter) {
  int res = mesh->volume(*iter, &ctxt, &volume);
  ...
  if (volume == 0.0)
    zero_volume_cells.push_back((*iter)->copy());
}
```

Testing for equality with floating-point numbers is always a little dubious, but we ignore that issue here. There are two things to note about the `push_back` statement. First, the extra parentheses around the `*iter` expression are necessary due to C++ precedence idiosyncrasies. Second, note that we call the `copy` method on the cell we get from dereferencing the iterator. *FM* iterators modify their cell values in place, for performance. We do not want another reference to the cell the iterator is using, we want a copy that will not change.

5.9 General Properties Interface

In Section 5.1 we listed the standard mesh and field properties that can be queried via dedicated access methods. Dedicated methods are convenient, efficient, and their use provides compile-time type checking. Unfortunately, an extensible design can never provide a complete set of dedicated property access functions. There are an arbitrarily large number properties that one may wish to associate with a mesh or field object; some mesh or field subclasses have properties that only apply to instances of those subclasses. Furthermore, various data file formats provide additional data about the data ("metadata") that would be appropriate to make available as properties. Through the `property_names` property described above, one can obtain the set of all property names associated with an object. The general `get` and `set` interface described below provides a means to access and modify the values.

FM provides two helper classes for representing general values: `Simple_Value<T>` and `Tuple_Value`. Their class declarations look like:

```
template <typename T>
class Simple_Value : public Object
{
public:
  Simple_Value(const T&);
  T get_value() const;
private:
  const T value;
};

class Tuple_Value : public Object
{
public:
  Tuple_Value(const std::vector<Ptr<Object> >&);
  Tuple_Value(Object*);
  Tuple_Value(Object*, Object*);
  size_t size() const;
  const Ptr<Object>& operator[](int) const;
private:
  std::vector<Ptr<Object> > values;
};
```

`Simple_Value<T>` is used for "simple" values such as scalars and strings. `Tuple_Value` is used for sequences of values. C++ pairs (`std::pair<S,T>`), sets

(`std::set<T>`) and vectors (`std::vector<T>`) are returned using tuple objects. *FM* `Vector<N,T>` instances are also returned as `N`-tuples. The `get` and `set` interface is declared as:

```
Ptr<Object> get(const std::string&,
                const Sub* =0, const Time* =0) const;
void set(const std::string&, const Ptr<Object>&,
         const Sub* =0, const Time* =0);
```

As with the dedicated property access functions, one can direct a request to a specific subblock or instance in time using the optional arguments. Calling `get` on a property that is not defined generates an exception. Call `set` on a property that is not defined adds that property to an object. Some care is required here: if one intends to set an extant property to a new value, but misspells the property name, then the object will silently add a new property under the misspelled name. Many properties are immutable; in particular, the properties listed in Table 5.1 cannot be changed. Calling `set` on those properties will at best be ignored, at worst it will lead to undefined behavior. Note that the `set` member function is not `const`; `set` is one of the few member functions that explicitly changes the state of a mesh or field object. The data structures used internally by *FM* field objects to store properties are protected by critical-section mutexes; calling `get` and `set` in multithreaded applications is safe. Though thread safe, using `set` with objects that are shared by multiple threads obviously should be done with care.

Finally, *FM* provides two access functions for cases where one wants to access arbitrary properties, and the property value type is known *a priori*:

```
template <typename T>
void get_simple_value(const std::string&, T*,
                      const Sub* =0, const Time* =0) const;
template <typename T>
void get_tuple_value(const std::string&, std::vector<T>*,
                     const Sub* =0, const Time* =0) const;
```

Attempting to access an undefined property, or a value not of the appropriate type, generates an exception. As a usage example, here is an alternate method for querying the physical dimensionality of a field:

```
unsigned d;
field->get_simple_value("phys_dimensionality", &d);
```

Using `get_phys_dimensionality` would be more efficient since there is no intermediate construction of a `Simple_Value<unsigned>` instance.

5.10 Miscellaneous

There are a few field methods that do not neatly fall neatly into the previously listed categories:

```
int volume(const Cell*, Context*, double*) const;
int n_at_cell(const Cell*, Context*, unsigned*) const;
int iblanks_at_cell(const Cell*, Context*, int*) const;
```

The `volume` method, as one would expect, computes the volume for a given cell. The `n_at_cell` method would typically be used in preparation for calling `at_cell`. (described in Section 5.6). In most applications, the number of values to be written into buffer provided as the final argument to `at_cell` is known in advance, thus one can use a fixed-size buffer. In the relatively infrequent cases where the number of values to be returned is not known *a priori*, an application can query what the number will be via the `n_at_cell` method; the value can then be used to dynamically allocate a buffer.

In the *FM* field interface design, we strove to be independent of any one file format standard. The `iblanks_at_cell` method is a concession to one particular file format: PLOT3D [32]. IBLANK values in PLOT3D serve two purposes: to flag nodes where the field values are "blanked", and to indicate overlap in multi-block meshes. IBLANK values are integers and are associated with the vertices of a mesh. Use of IBLANK values is discussed in more detail in the PLOT3D chapter.

Chapter 6

Structured Meshes

The mesh types that users are typically most familiar with are structured meshes. Regular meshes, rectilinear meshes and curvilinear meshes are all structured types. The base space for a structured mesh is defined as the Cartesian product of discretely sampled 1-D intervals. For a structured mesh with base dimensionality `B`, the Cartesian product defining the base space results in B-dimensional cubes and the faces of those cubes. For example, a 3-D base space is defined by the product of 3 intervals, and the mesh consists of hexahedral, quadrilateral, edge and vertex cells.

6.1 Structured Mesh Dimensions

The intervals used to define the base space are meshes themselves: the base and physical dimensionality for an interval is 1. The number of samples in an interval is equal to `card(0)`. The *dimensions* of a structured mesh consist of the number of samples for each axis in the Cartesian product. Dimensions are available as a mesh property, see Section 5.1. The `get_dimensions` method returns the dimensions as a `Vector<B,unsigned>` object. Traditionally, the axes of a structured mesh are labeled i, j, k, and so on. Let `dimensions` be the return value from `get_dimensions`, `dimensions[0]` corresponds to the i-axis, `dimensions[1]` to the j-axis, and so on.

The canonical enumeration (see Section 5.3) for structured meshes is defined with `dimensions[0]` (i-axis) changing most rapidly, followed by `dimensions[1]`, and so forth.

6.2 Simplicial Decomposition

Structured meshes with base dimensionality 2 or 3 optionally provide simplicial decomposition. When simplicial decomposition is turned on, quadrilaterals are split into triangles, and hexahedra are split into tetrahedra. For structured meshes the user has the choice of 3 simplicial decomposition modes. Mode 0 corresponds to no decomposition. Modes 1 and 2 specify decompositions where each hexahedron is broken into 5 tetrahedra. There are two 5-tetrahedra decompositions possible for a hexahedron. In order for

k-Cell	Alignment Bools			Alignment Dimensions			Alignment Cards		
k	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**
0	(f, f, f)			(7, 11, 13)			1001		
1	(t, f, f)	(f, t, f)	(f, f, t)	(6, 11, 13)	(7, 10, 13)	(7, 11, 12)	858	910	924
2	(t, t, f)	(t, f, t)	(f, t, t)	(6, 10, 13)	(6, 11, 12)	(7, 10, 12)	780	792	840
3	(t, t, t)			(6, 10, 12)			720		

Table 6.1: An example of the values in internal `Structured_Mesh<B,D>` data structures for a hexahedral mesh with dimensions (7, 11, 13). The **Alignment Bools** would be the same for any structured mesh with base dimensionality 3.

k-Cell	Cell Type	Alignment		
k		**0**	**1**	**2**
1	`EDGE_CELL`	i-edges	j-edges	k-edges
2	`QUADRILATERAL_CELL`	k-surface	j-surface	i-surface

Table 6.2: The names commonly associated with the standard alignments of a structured 3-mesh. A surface in a structured 3-mesh can be designated by fixing one of the axis indices to a specific value. For example, one specifies a k-surface by providing a specific value for the k axis.

the decompositions to be consistent between each pair of neighboring hexahedra, the decomposition for each hexahedron must be the opposite of its neighbors. Thus, given the decomposition choice for one hexahedron in a structured mesh, the choices for all the remaining hexahedra are forced. *FM* organizes the 2 decompositions in terms of "odd" and "even" vertices, where the odd and even designations come from the vertex indices. A vertex is even if the sum of its indices is even, otherwise it is odd. In decomposition mode 1, the diagonals added to decompose the quadrilaterals go between even vertices. The decomposition choices for the 6 quadrilateral faces of a hexahedron leave only one possible tetrahedral decomposition. In decomposition mode 2, the diagonals go between odd vertices, and the hexahedral decomposition follows suit.

6.3 Alignments

In *FM* one can instantiate a structured mesh with arbitrary base dimensionality B. A structured B-mesh contains k-cells, $0 \leq k \leq B$. We require a means to specify any particular cell. For the moment let us assume simplicial decomposition is off; k-cells, $k = 0..3$, correspond to vertices, edges, quadrilaterals, and hexahedra, respectively. In general, we would have k-cubes. In base space, a k-cell would extend along k different axes. A 0-cell would extend along no axes, a 1-cell along 1 axis, a 2-cell along 2 axes, and so on. We term the choice of which axes a cell is aligned with as an *alignment*. The number of possible alignments for a k-cell in a B-mesh can be expressed combinatorially: $\binom{B}{k}$, i.e., B choose k. For a hexahedral mesh, there is 1 alignment for vertices ($\binom{3}{0}$), 3 alignments for edges ($\binom{3}{1}$), 3 alignments for quadrialaterals ($\binom{3}{2}$), and 1 alignment for hexahedra ($\binom{3}{3}$). Table 6.1 provides a concrete example: the internal data

structure values for a mesh with dimensions (7, 11, 13). Under the **Alignment Bools** heading we see the boolean flags specifying along which axes cells with the given alignment extend. For example, a quadrilateral with alignment 0 extends along axes 0 and 1. The alignment bools are the same for any structured 3-mesh. The **Alignment Dimensions** are a function of the alignment bools and the structured mesh dimensions. Where the alignment bool is true, the alignment dimension is decremented by 1. Finally, **Alignment Cards** are defined as the product of the corresponding alignment dimensions. For example, for the mesh in Table 6.1, there are 858 edges aligned with axis 0. There are 2692 (858 + 910 + 924) edges total in the mesh. This is the same total one would get by calling `card(1)` on the mesh.

For structured 3-meshes, the terms i-surface, j-surface and k-surface are traditionally used to designate the standard surfaces produced by holding one axis index to a fixed value. Table 6.2 summarizes the correspondence between *FM* alignments and the traditional surface names.

6.4 Structured Cell Types

There are 5 structured cell types derived from `Structured_Cell<B>`. For casual users, it is probably not important to know all 5. Some cell types are implemented by more than one cell class. For example, there is more than one class that can represent a hexahedron. *FM* structured meshes may construct any of the 5 types when initializing an iterator (e.g, via `begin`), depending on the cell dimensionality and simplicial decomposition mode specified in the iterator attributes (see Section 6.5 below). The `begin` method is essentially a factory; extra effort is made within `begin` to choose the most optimal cell type for the requested iteration. The extra effort pays off during the iteration itself: specific cell classes provide small performance improvements that add up when making many `at_cell` calls.

6.4.1 `Structured_Cell<B>`

An abbreviated version of the structured cell declaration looks like:

```
template <int B>
class Structured_Cell : public Cell
{
public:
  const Vector<B,unsigned>& get_indices() const;
  const unsigned& operator[](int) const;
  virtual unsigned& operator[](int);
  virtual unsigned get_alignment() const;
  virtual unsigned get_subid() const;
protected:
  Vector<B,unsigned> indices;
};
```

The template parameter `B` specifies the base dimensionality of the mesh that the cell is intended to work with. The indices for any structured cell can be accessed via the

square brackets notation, e.g., `structured_cell[2] = 4`.

6.4.2 Structured_K_Cell<B>

`Structured_K_Cell<B>` is the most general class for representing k-cells in a structured B-mesh when simplicial decomposition is off. An appreviated version of the class declaration looks like:

```
template <int B>
class Structured_K_Cell : public Structured_Cell<B>
{
public:
  Structured_K_Cell(const Sub& s, const Time& t,
                    unsigned d, unsigned a,
                    const Vector<B,unsigned>& i);
private:
  unsigned dimension, alignment;
};
```

The concept of alignments is described previously in Section 6.3. `Structured_K_Cell<B>` can be used to represent any k-cell, $0 \leq k \leq B$. In cases where one knows *a priori* that k will be always be 0 or B, then one should use the classes dedicated to those cell types, which we describe next.

6.4.3 Structured_0_Cell<B>

The `Structured_0_Cell<B>` class represent vertices. There is no need to provide a dimension or alignment in a `Structured_0_Cell<B>` constructor.

6.4.4 Structured_B_Cell<B>

The `Structured_B_Cell<B>` class represents B-cubes, e.g., hexahedra in a 3-mesh, or quadrilaterals in a 2-mesh. As with `Structured_0_Cell<B>`, there is no need to provide a dimension or alignment argument in a `Structured_B_Cell<B>` constructor.

6.4.5 Structured_Subsimplex<B>

When simplicial decomposition is turned on, then we need objects that can represent subsimplexes within a B-cube.

```
template <int B>
class Structured_Subsimplex : public Structured_Cell<B>
{
public:
  Structured_Subsimplex(const Sub& s, const Time& t,
                        unsigned d, unsigned sid,
                        const Vector<B,unsigned>&);
private:
  unsigned dimension, subid;
};
```

Attribute	Arg. Types	Default
ALIGNMENT	unsigned	*none*
AXIS_BEGIN	unsigned axis, unsigned i	0
AXIS_END	unsigned axis, unsigned i	*see caption*
AXIS_STRIDE	unsigned axis, unsigned i	1
CELL_DIMENSION	unsigned	0
CELL_TYPE	int	VERTEX_CELL
I_SURFACE	unsigned	*none*
J_SURFACE	unsigned	*none*
K_SURFACE	unsigned	*none*
SIMPLICIAL_DECOMPOSITION	unsigned	0
SUB	Sub	*Sub undefined*
TIME	Time	*Time undefined*

Table 6.3: The standard structured mesh `Iter_Attr` codes. The default `AXIS_END` values depend on the cell type and whether an alignment is specified.

6.4.6 `Structured_B_Subsimplex<B>`

Analogous to the performance-optimized class for a B-cube, we have an optimized class for a structured B-subsimplex, i.e., a subtetrahedron of a hexahedron, or the subtriangle of quadrilateral.

6.5 Iterators

In Section 5.8 we introduced the iterator interface for *Field Model* meshes and fields. By default *FM* iterators loop over every vertex in a mesh. To change the set or type of cells iterated over, one can provide an `Iter_Attrs` argument to `begin` or `split_begin`. Table 6.3 summarizes the `Iter_Attr` keywords for structured meshes. Typical usage looks like:

```
Iter_Attrs iter_attrs;
iter_attrs.push_back(Iter_Attr(K_SURFACE, 2));
iter_attrs.push_back(Iter_Attr(AXIS_BEGIN, 0, 5));
iter_attrs.push_back(Iter_Attr(AXIS_END, 0, 10));
for (Iter iter = field->begin(&iter_attrs); !iter.done(); ++iter) {
  int res = field->get_mesh()->at_cell(*iter, &ctxt, xyzs);
  ...
  res = field->at_cell(*iter, &ctxt, vals);
  ...
}
```

We walk through the `Iter_Attr` options for structured behavior objects next.

6.5.1 `ALIGNMENT`

By default, an iterator will iterate over every alignment that is appropriate for a given cell type. The k-cells in a B-mesh $\mathcal{M}$ have more than one alignment if $0 < k < B$, in

other words, if the cell type is neither a vertex nor a B-cube in $\mathcal{M}$. Via the `ALIGNMENT Iter_Attr`, one can restrict the iteration to a specific alignment.

6.5.2 AXIS_BEGIN, AXIS_END, AXIS_STRIDE

The loop construct for an axis, given `begin`, `end` and `stride` values, would look like:

```
for (unsigned i = begin; i < end; i += stride)
  ...
```

For a B-mesh, the loops for the B axes would essentially be nested, with axis 0 as the innermost loop. The `AXIS_BEGIN` and `AXIS_STRIDE` values for each axis default to 0 and 1, respectively. The `AXIS_END` default value depends on the alignment implied by the iterator attributes. Recall in Section 6.3 we introduced the concepts of alignments and alignment dimensions. Table 6.1 provided a concrete example. The alignment dimensions provide the default iteration end values for a particular cell type and alignment.

Recall that k-cells, $0 < k < B$, have multiple alignments. For such cells, if the `ALIGNMENT` attribute is not given in the initialization, then the iterator will loop over each alignment. This implies an ambiguity: which alignment should `AXIS_BEGIN`, `AXIS_STRIDE` and `AXIS_END` attributes be applied to? Applying the attributes to all alignments is one possibility, but this interpretation is not always meaningful. Currently the implementation takes the more conservative approach: the `AXIS_BEGIN`, `AXIS_STRIDE` and `AXIS_END` attributes are ignored multi-alignment cases. If necessary one can still control these attributes by iterating over one alignment at a time.

6.5.3 CELL_DIMENSION, CELL_TYPE

One can directly specify the type of cells to iterate over via either `CELL_DIMENSION` or `CELL_TYPE`. If both dimension and type are specified, then cell type trumps cell dimension. If the cell type is given as `TRIANGLE_CELL` or `TETRAHDRON_CELL` and simplicial decomposition mode is 0 (either because it was not specified or because it was explicitly given as 0) then the simplicial decomposition mode is set to 1.

6.5.4 I_SURFACE, J_SURFACE, K_SURFACE

The `I_SURFACE`, `J_SURFACE` and `K_SURFACE` attributes provide a convenient shorthand in 3-mesh cases for initializing an iterator to loop over the 2-cells in a surface defined by holding one of the structured mesh indices constant. The alignment is implied by the `I`, `J`, or `K` designation. See Table 6.2

6.5.5 SIMPLICIAL_DECOMPOSITION

The `SIMPLICIAL_DECOMPOSITION` option is available for meshes with base dimensionality 2 or 3. See Section 6.2.

6.5.6 SUB, TIME

By default in multi-block cases an iterator will iterate over all the subblocks. If a `SUB Iter_Attr` is provided, then the iteration will be over the particular subblock only. For more on multi-block iteration, see Chapter **??**

For time-varying data, time must be set on cell arguments, including the cell arguments provided by dereferenced iterators. Time is set via the `TIME` iterator attribute:

```
Iter_Attrs iter_attrs;
iter_attrs.push_back(Iter_Attr(SUB, Sub(10)));
iter_attrs.push_back(Iter_Attr(TIME, Phys_Time(3.0)));
```

It is difficult to set the subblock or time on an iterator once it is initialized because the dereferenced iterator type is `const Cell*` and the `Cell` `set_sub` and `set_time` methods are not `const`.

6.6 Regular_Interval

`Regular_Interval` is derived from `Structured_Mesh<1,1>`, it represents a 1-D interval. The constructor looks like:

```
Regular_Interval::
Regular_Interval(unsigned n,
                 Coord origin = Coord(0), Coord delta = Coord(1));
```

The `delta` value is required to be non-zero. Negative values for `delta` are allowed. `Regular_Interval` is typically used in the construction of higher-dimensional objects. See for example `Product_Mesh<B,D>`.

6.7 Irregular_Interval

`Irregular_Interval` is derived from `Structured_Mesh<1,1>`, just as with regular intervals. The constructor looks like:

```
Irregular_Interval::
Irregular_Interval(unsigned n, const Coord* x);
```

The coordinate values `x` are required to be strictly ascending or descending. As with regular intervals, `Irregular_Interval` is typically used in the construction of product meshes. The user can control whether an `Irregular_Interval` instance will deallocate its coordinate buffer when it is destructed via the `delete_suppression` property. See Section 3.7.

6.8 Product_Mesh<B,D>

The `Product_Mesh<B,D>` class represents rectilinear meshes aligned with the physical coordinate axes. The meshes can be defined as the Cartesian product of B 1-D axes. Those axes are passed in to the `Product_Mesh<B,D>` constructor:

Cylindrical Centerline	Physical Axis Indices		
	Radial	Rotational	Length
X-Axis	1	2	0
Y-Axis	2	0	1
Z-Axis	0	1	2

Table 6.4: The standard choices for the physical axis indices in a `Cylindrical_Mesh` constructor. By default the cylinder centerline is the Z-axis. The standard combinations produce right-handed cells.

```
template <int B, int D>
Product_Mesh<B,D>::
Product_Mesh(const std::vector<Ptr<Mesh<1,1> > >& axes);
```

The `Mesh<1,1>` objects used to specify each axis are typically instances of `Regular_Interval` or `Irregular_Interval`, described above. `Product_Mesh<B,D>` instances have structured behavior, and each of the axes must also have structured behavior. In the cases where $B < D$, the $D - B$ highest physical coordinates are fixed at 0.0. For example, `Product_Mesh<2,3>` would represent a rectangular surface mesh in the $Z = 0$ plane.

6.9 `Regular_Mesh<B,D>`

`Regular_Mesh<B,D>` objects are essentially `Product_Mesh<B,D>` instances with more convenient constructors and some performance optimizations. Both `Regular_Mesh<B,D>` constructors take the mesh dimensions as the first argument. By default the origin and spacing along each axis is 0 and 1, respectively. The second form of the `Regular_Mesh<B,D>` constructor takes arguments enabling one to specify an origin and a spacing different from the default. The two constructor declarations are:

```
template <int B, int D>
Regular_Mesh<B,D>::
Regular_Mesh(const Vector<B,unsigned>& dimensions);

template <int B, int D>
Regular_Mesh<B,D>::
Regular_Mesh(const Vector<B,unsigned>& dimensions,
             const Vector<B,double>& origin,
             const Vector<B,double>& spacing);
```

The implementation is free to provide `Regular_Mesh<B,D>` specializations that are optimized for various dimensionalities. Currently the implementation provides a full specialization for some of the methods of the most popular instantiation type: `Regular_Mesh<3,3>`. See `FM_Regular_Mesh.[hC]`.

6.10 Cylindrical_Mesh

The Cylindrical_Mesh class represents cylindrical meshes aligned with the physical coordinate axes. Both the base and physical dimensionality for Cylindrical_Mesh objects are 3. Cylindrical_Mesh are defined such that the radial axis extends in the direction of 0 rotation, the rotational axis extends in the direction of one quarter revolution, and the length axis extends in the direction of the cylindrical centerline. The Cylindrical_Mesh constructor definition is:

```
Cylindrical_Mesh::
Cylindrical_Mesh(const Ptr<Mesh<1,1> >& radial_axis,
                 const Ptr<Mesh<1,1> >& rotational_axis,
                 const Ptr<Mesh<1,1> >& length_axis,
                 unsigned phys_radial_axis_index = 0,
                 unsigned phys_rotational_axis_index = 1,
                 unsigned phys_length_axis_index = 2,
                 unsigned base_radial_axis_index = 0,
                 unsigned base_rotational_axis_index = 1,
                 unsigned base_length_axis_index = 2);
```

The first 3 arguments to the constructor are 1-D meshes specifying the coordinates to use for each axis. Rotational coordinates should be in radians. Each of the 3 axes is required to have structured behavior. The next three arguments specify which axis the cylinder is aligned with. Table 6.4 summarizes the standard choices. By default the centerline of the cylinder is the Z-axis. One can imagine 6 possible combinations, 3 beyond those listed in Table 6.4. The remaining three combinations would define cylindrical meshes with left-handed cells. The constructor detects this; using combinations that result in left-handed cells is OK. The final three arguments specify the pairings between the base and physical axes.

The Cylindrical_Mesh specification is intended to be flexible enough to represent a wide variety of axis-aligned cylinder uses. Compared to the more general Curvilinear_Mesh<3,3>, Cylindrical_Mesh objects require less memory and have better point location performance.

6.11 Curvilinear_Mesh<B,D>

Curvilinear meshes are the most general type of structured mesh in *FM*. Curvilinear_Mesh<B,D> is an abstract class and parent to derived classes distinguished by how the coordinates are stored. We review those classes next.

6.11.1 Curvilinear_Mesh_T_Layout<B,D,U>

The class Curvilinear_Mesh_T_Layout<B,D,U> is constructed with a single array of Vector<D,Coord>, the type "T" in T_Layout. The constructor looks like:

```
template <int B, int D, typename U = unsigned>
Curvilinear_Mesh_T_Layout<B,D,U>::
Curvilinear_Mesh_T_Layout(const Vector<B,unsigned>& dimensions,
                          const Vector<D,Coord>* coordinates);
```

The final template argument `U` specifies the type to use for array indexing. The type should be either `unsigned` (the default) or `unsigned long long`. Typically the `unsigned` type consists of 32 bits and `unsigned long long` consists of 64. Use of 64-bit values adds a small cost to indexing calculations. Since coordinate access tends to be one of the most performance-critical operations, we provide the choice of indexing type.

The convention in *FM* for mesh and field constructors is that any buffer provided as a constructor argument becomes the reponsibility of the *FM* object to deallocate. One can suppress the default deallocation behavior by setting the `delete_suppression` property to `true`. See Section 3.7.

Chapter 7

Unstructured Meshes

The unstructured mesh classes in *Field Model* inherit from the parent abstract class `Unstructured_Mesh<B,D>`. There are several classes to choose from, each distinguished by the internal data structures used to represent the mesh. We review the current classes next.

7.1 `Unstructured_Vertex_Mesh<D>`

`Unstructured_Vertex_Mesh<D>` is designed for meshes with a base dimensionality of 0, i.e., what are known as "scattered vertex" meshes. The constructor looks like:

```
template <int D>
Unstructured_Vertex_Mesh::
Unstructured_Vertex_Mesh(unsigned n, const Vector<D,Coord>* x);
```

The class is templated on the physical dimensionality D, typical instantiation values for D are 2 or 3.. In Figure 2.1, the $(0, 2)$ and $(0, 3)$ cases correspond to `Unstructured_Vertex_Mesh<2>` and `Unstructured_Vertex_Mesh<3>` instances, respectively.

7.2 `Unstructured_Edge_Mesh<D>`

`Unstructured_Edge_Mesh<D>` is designed for meshes with a base dimensionality of 1, i.e., meshes consisting of vertices and edges. The constructor looks like:

```
template <int D>
Unstructured_Edge_Mesh<D>::
Unstructured_Edge_Mesh(unsigned n_vertices, const Vector<D,Coord>* x,
                       unsigned n_edges, const Vector2u* edges);
```

The class is templated on the physical dimensionality D. As with the unstructured vertex mesh class described above, typical instantiation values are 2 or 3. In Figure 2.1, the $(1, 2)$ and $(1, 3)$ cases correspond to `Unstructured_Edge_Mesh<2>` and

`Unstructured_Edge_Mesh<3>` instances, respectively. Note that the class is constructued with minimal connectivity information: the `edges` array simply consists of pairs of unsigned integers specifying the indices for the two vertex faces of each edge. `Unstructured_Edge_Mesh<D>` does not construct additional connectivity data structures, thus some topological requests can be relatively expensive to compute. In particular, for a mesh with e edges, the 1-star of a vertex is computed in order $O(e)$ time.

7.3 Unstructured_Triangle_Mesh<D>

`Unstructured_Triangle_Mesh<D>` is designed for meshes consisting of triangles, edges and vertices. Typical values for D are 2 and 3. The constructor looks like:

```
template <int D>
Unstructured_Triangle_Mesh<D>::
Unstructured_Triangle_Mesh(unsigned n_v, const Vector<D,Coord>* x,
                           unsigned n_t, const Vector3u* t);
```

Like `Unstructured_Edge_Mesh<D>`, `Unstructured_Triangle_Mesh<D>` is constructed with the coordinates of each vertex and minimal connectivity information: the vertex indices for each triangle for the triangle mesh case. Unlike the unstructured edge mesh class, `Unstructured_Triangle_Mesh<D>` may construct additional internal data structures if required by the access requests. In particular, `Unstructured_Triangle_Mesh<D>` instances can enumerate the edges in the mesh if needed. Requests that require edge enumeration include `card(1)`, topological operations involving 1-cells, and iteration over 1-cells.

7.4 Unstructured_TPWH_Mesh

The `Unstructured_TPWH_Mesh` class represents meshes with base and physical dimensionality of 3. The 3-cells are tetrahedra, pyramids, prisms, and hexahedra. The `TPWH` designation comes from the 3-cell names, prisms are also known as “wedges”. There are two constructors:

```
Unstructured_TPWH_Mesh::
Unstructured_TPWH_Mesh(unsigned n_vertices, const Vector3C* xyzs,
                       unsigned n_tetrahedra, Vector4u* tetrahedra,
                       unsigned n_pyramids, Vector5u* pyramids,
                       unsigned n_prisms, Vector6u* prisms,
                       unsigned n_hexahedra, Vector8u* hexahedra);

Unstructured_TPWH_Mesh::
Unstructured_TPWH_Mesh(unsigned n_vertices, const Vector3C* xyzs,
                       unsigned n_tetrahedra, Vector4u* tetrahedra,
                       unsigned n_pyramids, Vector5u* pyramids,
                       unsigned n_prisms, Vector6u* prisms,
                       unsigned n_hexahedra, Vector8u* hexahedra,
                       Triangle_Vertices_To_Facet_Info_Map*,
                       Quadrilateral_Vertices_To_Facet_Info_Map*);
```

The second constructor includes two extra arguments: a pointer to a hash table that maps from triangle vertices to facet info, and a pointer to a hash table that maps from quadrilateral vertices to facet info. Facets are 2-cells (triangles and quadrilaterals here), and facet info is a data structure containing information about the 1 or 2 3-cells that each facet is the face of. Facet info is used for `neighbors` calls and point location algorithms that walk from one 3-cell to another.

Bibliography

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

[2] A. Alexandrescu. *Modern C++ Design*. Addison Wesley, 2001.

[3] P. Alexandroff. *Elementary Concepts of Topology*. Dover Publications, Inc., New York, 1961. Translated by Alan E. Farley.

[4] P. S. Alexandroff. *Combinatorial Topology*. Dover Publications, Inc., New York, 1998.

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

[6] G. Bancroft et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings of Visualization '90*, pages 14–24. IEEE Computer Society Press, October 1990.

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

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

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

[10] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[11] FITS. `http://www.gsfc.nasa.gov/astro/fits/fits_home.html`.

[12] National Center for Supercomputing Applications. Hierarchical Data Format. `http://hdf.ncsa.uiuc.edu/`.

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

[14] S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *Computing in Science & Engineering*, 1(4):66–72, Jul/Aug 1999.

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

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

[17] D. Kenwright. Automatic detection of open and closed separation and attachment lines. In *Proceedings of Visualization 1998*, pages 151–158. IEEE Computer Society Press, October 1995. using delta wing data, not time-varying.

[18] D. Kenwright and D. Lane. Optimization of time-dependent particle tracing using tetrahedral decomposition. In *Proceedings of Visualization '95*. IEEE Computer Society Press, October 1995.

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

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

[21] S. Meyers. *More Effective C++*. Addison Wesley, 1996.

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

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

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

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

[26] NCSA/NASA. HDF-EOS. `http://hdfeos.gsfc.nasa.gov/`.

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

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

[29] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition, 2000.

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

[31] T. Veldhuizen. Expression templates. *C++ Report*, 7(26), 1995.

[32] P. Walatka, P. Buning, L. Pierce, and P. Elson. *PLOT3D User's Manual*. National Aeronautics and Space Administration, July 1992. NASA Technical Memorandum 101067.