New iMesh paper draft

Draft of April 2, 2008

1 Introduction

Creating simulation software for problems described by partial differential equations is a relatively common but very time-consuming task. Much of the effort of developing a new simulation code goes into writing infrastructure for tasks such as interacting with mesh and geometry data, equation discretization, adaptive refinement, design optimization, etc. Because these infrastructure components are common to most or all simulations, re-usable software for these tasks would significantly reduce both the time and expertise required to create a new simulation code.

Currently, libraries are the most common mechanism for software re-use in scientific computing, especially the highly-successful libraries for numerical linear algebra [**?**, **?**, **?**, **?**, **?**]. The drawback to software re-use through libraries is the difficulty in changing from one to another. When a user wishes to add functionality or simply experiment with a different implementation of the same functionality in another library, all calls within an application must be changed to the other API. To make matters worse, different libraries rarely package functionality in precisely the same way. Another significant challenge with library use, especially in the context of meshing and geometry libraries, is that data structures used within the libraries may be radically different, making changes from one library to another even more onerous. This time-consuming conversion process can be a significant diversion from the central scientific investigation, so many application researchers are reluctant to undertake it. As a result, new advances from the meshing research community often take years to become incorporated into application simulations.

Components represent a higher level of abstraction than libraries: essentially, a component defines both a standard application programming interface (API) for the functionality provided by a collection of analogous libraries and an abstract data model defining what sorts of data are passed through the interface. Returning to the familiar example of linear

algebra, a numerical linear algebra component would define a standard interface for operations like dot products, matrix-vector multiplication, and linear system solution, and its abstract data model would include objects like vectors and matrices. A key advantage to the use of components is that any service that uses the component API can interact with any implementation of the API, including partial implementations designed to provide only the functionality required by that service; we will return to this point in Sections 1.1 and 5.

This paper will describe a meshing component intended to support low-level mesh access and manipulation. In addition, our interface is designed to support the requirements of solver applications, including the ability to define mesh subsets and to attach arbitrary user data to mesh entities. Also, our component is intended to be both language and data structure independent. In summary, our initial target is to support low-level interaction between applications programs — both meshing and solution applications — and external mesh databases regardless of the data structures and programming language used by each.

The most prominent example of prior research in defining interfaces for meshing is the Unstructured Grid Consortium, a working group of the AIAA Meshing, Visualization, and Computing Environments Technical Committee [**?**]. The first release of the UGC interface [**?**] was aimed at high level mesh operations, including mesh generation and quality assessment. Recognizing a need for lower-level functionality, the UGC has developed a low-level query and modification interface for mesh databases aimed exclusively at meshing operations, as well as an interface for defining generic high-level services [**?**]; results of such queries in the UGC interface are explicitly expressed as integer indices into data arrays. The low-level UGC interface is similar in scope to our API, although we have deliberately been more general in providing support for functionality required by solvers and in emphasizing data structure neutrality.

1.1 A Simple Use Case for a Meshing Component

Before we begin general discussion of the meshing component we have developed, let us first consider a simple example illustrating how a scientific computing application could benefit from using such a component. As an example of a typical scientific computing application, let us consider a finite element solver (FESolve) for some partial differential equation, and how this application might continue to develop over time. [1] When first written, FESolve is a simple finite volume solver, using linear elements. At runtime, FESolve loads a mesh from a file and does some pre-processing of the mesh to compute geometric quantities (such as integration points and weights) and perhaps to compute some mesh

[1] While different applications will surely have different requirements for interacting with unstructured mesh data, many, if not most, applications will follow roughly this same outline.

topological relationships that weren't in the file. Then, for each iteration, FESolve iterates over the elements in the mesh, computing the residual and the stiffness matrix for each, and assembling these into a global linear system. This system is solved, and the solution is updated at every node.

After FESolve has been in use for some time, its developers decide that mesh adaptation is required to improve solution accuracy. With current approaches to writing mesh infrastructure software, they now face three choices: either to write their own mesh adaptation code; to change the data structures in FESolve so that they can use an existing mesh adaptation code written by some other researcher; or to have FESolve communicate with an external mesh adapter through files. None of these approaches is ideal from the developers' point of view, but eventually they pick one and start in.

Using a standard mesh component API makes this transition much less painful. The developers choose mesh adaptation code that uses the component API to access mesh data. They then add to FESolve implementations of *only* the functions in the API *actually required* for mesh adaptation. Now their data, in their own internal data structures, can be used directly by the mesh adaptation code without further integration.

As a bonus, in implementing part of the mesh component's API, the FESolve development team has simultaneously done some of the work that will be required when their users start clamoring for other advanced capability (design optimization, for instance) which could also be provided using the mesh component API.

1.2 The ITAPS Mesh Component

In this paper, we will describe a newly developed component intended to provide support for the mesh access and manipulation requirements of practical, large-scale scientific computing applications. This component, developed as part of a larger project by the Interoperable Tools for Advanced Petascale Simulation (ITAPS) center to develop interoperable software tools for meshes, domain geometry, and solution representation [**?**], is called iMesh. Note the words "support for": the iMesh component is not intended to be a general interface to all possible meshing operations, but rather, to define the operations required at a mesh database level so that high-level operations — including mesh generation, mesh improvement, mesh adaptation, and design optimization — can be implemented as *services* that use the iMesh component to store and manipulate mesh data. To be genuinely useful to real applications and real application developers, the component must be

- general purpose: all mesh operations must be implementable based on the iMesh component.

- efficient: data access using the iMesh component must not come at too high a cost in overhead.
- flexible: different applications may want to use different approaches for the same task.
- interoperable: implementations of the component must be truly interchangeable, and services designed to use the interface should work on a plug and play basis, regardless of data structures and programming language.

Section 2 describes the design principles we followed to ensure that the iMesh component met these goals. We first defined a data model (see Section 3): meshing operations require information about mesh entities (like vertices, triangular faces, and hexahedral regions), collections of entities, and meta-data associated with mesh entities. Using that data model, we then defined an API that would support general meshing and mesh-related solver operations (see Section 4). In addition to defining the iMesh component, we have also used it for various meshing and PDE solution tasks; several examples will be given in Section 5. The paper will conclude with discussion of lessons learned from developing this component, of the current status of software using the iMesh API, and of future prospects for extension and application of the iMesh component.

2 Design Principles

In Section 1.2, we summarized our goals for the iMesh component. As design of the component continued, we found that several principles recurred frequently in guiding our design decisions as we worked towards those goals. Specifically, we found that we made decisions to produce an interface that was:

Complete. Clearly, a minimal requirement is that all required mesh operations must be possible, either intrinsically through the iMesh API or by building on it. Our guide to whether the interface is complete has been our collective experience — from quite different perspectives — in meshing and solution algorithms. We feel that if we can satisfy our collective needs, and those that we know are in use by others, that the interface is complete enough for at least an initial release.

Run-time efficiency. For the iMesh component to be useful for applications it must have low overhead. Specifically, the interface must be designed so that an iMesh implementation can provide data access and manipulation nearly as rapidly as native access to the same mesh database. An example of the application of this principle

in the iMesh interface are the availability of both single-entity and array-of-entities access to mesh data, each of which is more efficient under some but not all circumstances.

Ease of use. We also wish to design an interface which is a relatively easy for programmers to use. This implies an interface that is relatively compact on the one hand but also provides direct access to relatively commonly used constructs, even at the expense of additional functions in the interface. For example, we recognize that certain types of metadata – specifically, double, integer, and entity handle metadata – will be very common and more easily handled both by iMesh implementations and applications if there are specific functions for these types. However, in addition to these particular functions, we also provide in such cases a general access mechanism; in this case, generic metadata is described using byte strings.

Flexibility. We recognize that different applications may choose to express the same semantic content in different ways. Where feasible, the iMesh interface supports this. For example, one application may choose to represent boundary condition data by metadata attached to particular mesh entities; another may represent the same information by collecting entities with the same boundary condition into a set. As another example, some applications may choose to access data entity by entity while others may prefer array access to data.

Extensibility. We have designed the interface to allow extensions to the low-level mesh access functionality that interface defines. For example, a recent addition to the iMesh interface is support for nonlinear shape of mesh entities. The support was added to the iMesh interface without requiring changes to functions already in the interface. As a second example, ongoing work for it parallel extension to the iMesh interface leverages serial iMesh functionality for parallel usage where appropriate.

Simplified applications programming. One obvious way to simplify applications programming is by making the iMesh interface as lightweight as possible. In addition, wherever possible, the iMesh interface is designed to place difficult tasks under the control of the implementation rather than the application. A prime example of this is in the area of memory management. An application, when requesting an array of data, need not know in advance the size of the array. Instead, the application can pass in an uninitialized array and implementation automatically allocates the appropriate amount of memory for that array.

Interoperability. In the long-term, success of the iMesh component will depend on how well the component truly supports interoperability. This is the key to being able

to leverage the effort in development of both implementations and services as well as conversion of applications to use the interface. Interoperability, in turn, requires not only the use of a standard interface, but also data structure and programming language neutrality.

3 Data Model

In the iMesh data model, all mesh primitives — vertices (0D), edges (1D), faces (2D), and regions (3D) — are referred to as *entities*. Mesh entities are collected together to form *entity sets*. All topological and geometric mesh data[2] is stored in a *root entity set* and there is a single root set for each computational domain; all other entity sets are contained in the root set. Many implementations will represent the root set as a database containing all of the mesh entities, with other entity sets containing handles for these entities. Any iMesh data object — an entity or any entity set including the root set — can have one or more *tags* associated with it, so that arbitrary data can be attached to the object. To preserve data structure neutrality, all iMesh data objects are identified by opaque handles.

3.1 Mesh Entities

All the primitive components of a mesh are defined by the iMesh data model to *entities*. iMesh entities are distinguished by their entity type (effectively, their topological dimension) and entity topology; each topology has a unique entity type associated with it. Examples of entities include a vertex, an edge, triangular or quadrilateral faces in 2D or 3D, and tetrahedral or hexahedral regions in 3D; a complete catalog of entities supported by iMesh is shown in Figure 1. Faces and regions have no interior holes. Higher-dimensional entities are defined by lower-dimensional entities using a canonical ordering.

3.2 Entity Adjacencies

Adjacencies describe how mesh entities connect to each other. For an entity of dimension d, first-order adjacency returns all of the mesh entities of dimension q which are on the closure of the entity for downward adjacency ($d > q$), or for which the entity is part of the closure for upward adjacency ($d < q$), as shown in Figure 2(a) and (b). For a particular implementation, not all first-order adjacencies are necessarily available. For instance, in a classic finite element element-node connectivity storage, requests for faces or edges

[2] *Geometric mesh data* is geometric data required to define shapes of mesh entities. This is distinct from *geometric model data*, which defines the shapes of the problem domain.

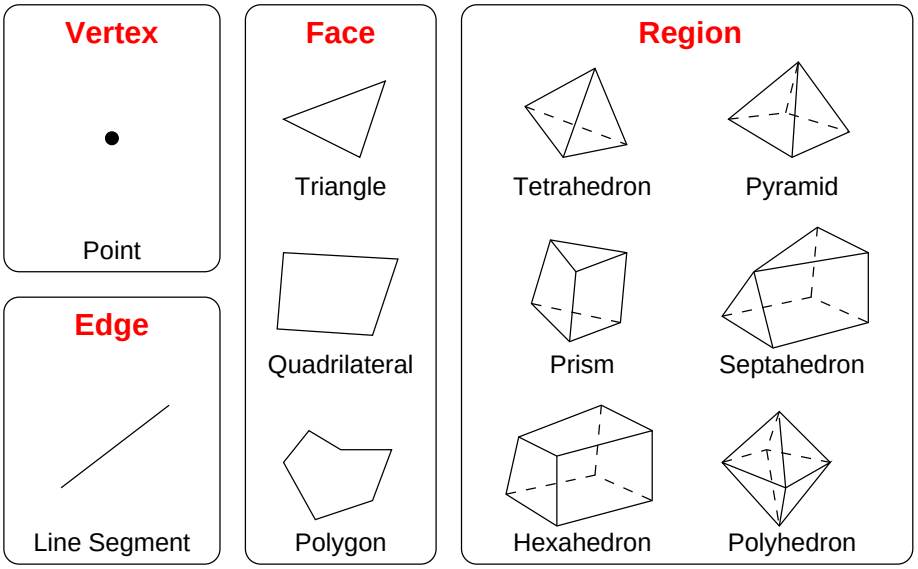
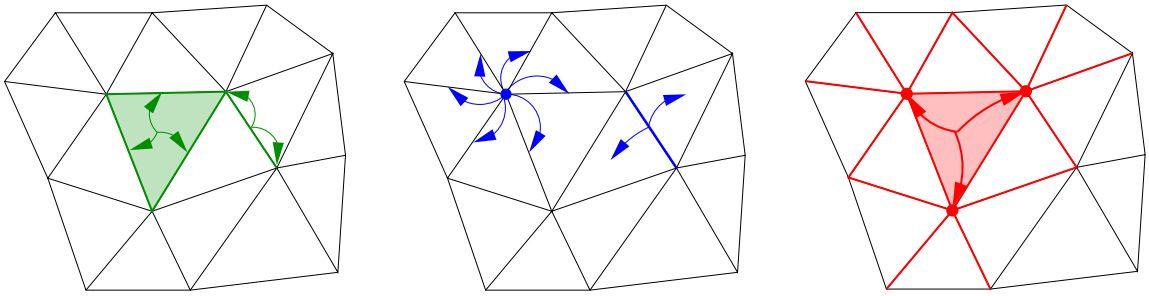


Figure 1: Entities supported by the iMesh component, including the ordering of the lower-dimensional entities on their closure.

adjacent to an entity may return nothing, because the implementation has no stored data to return. For first-order adjacencies that are available in the implementation, the implementation may store the adjacency information directly, or compute adjacencies by either a local traversal of the entity's neighborhood or by global traversal of the entity set. Each iMesh implementation must provide information about the availability of and relative cost of first-order adjacency queries.

For an entity of dimension d, second-order adjacencies describe all of the mesh entities of dimension q that share any adjacent entities of dimension b, where $d \neq b$ and $b \neq q$. Second-order adjacencies can be derived from first-order adjacencies. Note that, in the iMesh data model, requests such as all vertices that are neighbors to a given vertex are requests for second-order adjacencies.

Examples of adjacency requests include: for a given face, the regions on either side of the face (first-order upward); the vertices bounding the face (first-order downward); and the faces that share any vertex with the face (second-order).



(a) Downward adjacency; edges adjacent to a face, vertices adjacent to an edge.

(b) Upward adjacency: edges adjacent to a vertex, faces adjacent to an edge.

(c) Second adjacency; red edges are adjacent to vertices adjacent to the red face.

Figure 2: Examples of adjacency relationships between mesh entities.

3.3 Entity Sets

The iMesh data model includes the notion of arbitrary groupings of entities; these groupings are called *entity sets*. Each entity set may be a true set (in the set theoretic sense) or it may be a (possibly non-unique) ordered list of entities; in the latter case, entities are retrieved in the order in which they were added to the entity set. An entity set also may or may not be a simple mesh; entity sets that *are* simple meshes have obvious application in multiblock and multigrid contexts, for instance. Entity sets (other than the root set) are populated by addition or removal of entities from the set. In addition, set boolean operations — subtraction, intersection, and union — are also supported.

Two primary relationships among entity sets are supported. First, entity sets may contain one or more entity sets (by definition, all entity sets belong to the root set). An entity set contained in another may be either a subset or an element (each in the set theoretic sense) of that entity set. The choice between these two interpretations is left to the application; the iMesh interface does not impose either interpretation. Set contents can be queried recursively or non-recursively; in the former case, if entity set A is contained in entity set B, a request for the contents of B will include the entities in A (and the entities in sets contained in A). Second, parent/child relationships between entity sets are used to represent logical relationships between sets, including multigrid and adaptive mesh sequences. These logical relationships naturally form a directed, acyclic graph.

Examples of entity sets include the ordered list of vertices bounding a geometric face, the set of all mesh faces classified on that geometric face, the set of regions assigned to a single processor by mesh partitioning, and the set of all entities in a given level of a multigrid mesh sequence.

3.4 Tags

Tags are used as containers for user-defined data that can be attached to iMesh entities, meshes, and entity sets. Different values of a particular tag can be associated with different mesh entities; for instance, a boundary condition tag will have different values for an inflow boundary than for a no-slip wall. In the general case, iMesh tags do not have a predefined type and allow the user to attach arbitrary data to mesh entities; this data is stored and retrieved by implementations as a bit pattern. To improve performance and ease of use, we support three specialized tag types: integers, doubles, and handles. These typed tags enable correct saving and restoring of tag data when a mesh is written to a file.

3.5 Meshes

To be useful to applications, information in the root set or one or more of its constituent entity sets is assumed to be a valid computational mesh, examples of which include:

- A non-overlapping, connected set of iMesh entities; for example, the structured and unstructured meshes commonly used in finite element simulations (*simple mesh*).
- Overlapping grids in which a collection of simple meshes are used to represent some portion of the computational domain, including chimera, multiblock, and multigrid meshes (*multiple mesh*). The interfaces presented here handle these mesh types in a general way; higher-level convenience functions may be added later to support specific functionalities needed by these meshes. In this case, each of the simple meshes is a valid computational mesh, stored as an entity set.
- Adaptive meshes in which all entities in a sequence of refined (simple or multiple) meshes are retained in the root set. The most highly refined adaptation level typically comprises a simple or multiple mesh. Typically, different levels of mesh adaptation will be represented by different entity sets, with many of the entities shared by multiple entity sets.
- Smooth particle hydrodynamic (SPH) meshes, which consist of a collection of iMesh vertices with no connectivity or adjacency information.

At the most fundamental level, we consider a static simple mesh. This mesh provides only basic query capabilities to return entities and their adjacencies. This implies that all implementations have a root set, but not necessarily the subsetting capabilities described in Section 3.3.3.

In addition, meshes can also be extended to be modifiable, through support for creation and deletion of mesh entities (see Section **??**.4.3). Modifiable meshes require a minimal interaction with the underlying geometric model to uniquely associate mesh entities with geometric model entities of equal or greater dimension[**?**].

4 Interface Functionality

The iMesh interface supports a variety of commonly needed functionalities for mesh and entity query, mesh modification, entity set operations, and tags. All data passed through the interface is in the form of opaque handles to objects defined in the data model. In this section we describe the functionality available through the iMesh interface.[3] For listings of allowable values of all ITAPS enumerated data types and a concrete example of the full call sequence for functions with arrays and strings, see Appendix **??**.

4.1 Global Queries

Global query functions can be categorized into two groups: 1) *database functions*, that manipulate the properties of the database as a whole and 2) *set query functions*, that query the contents of entity sets as a whole; these functions require an entity set argument, which may be the root set as a special case. These functions are summarized in Table 1.

Database functions include functions to create and destroy mesh instances; note that the create function only sets up data structures for the mesh instance, without supplying any mesh data. The load and save functions read and write mesh information from files; file format and read/write options are implementation dependent. As mesh data is loaded, entities are stored in the root set, and can optionally be placed into a subsidiary entity set as well. iMesh implementations must be able to provide coordinate information in both blocked (xxx...yyy...zzz...) and interleaved (xyzxyzxyz...) formats; an application can query the implementation to determine the implementation's preferred storage order. Also, implementations must provide information about the availability and relative cost of computing adjacencies between entities of different types. Finally, each instance of the interface must provide a handle for the root set.

Set query functions allow an application to retrieve information about entities in a set. The entity set may be the root set, which will return selected contents of the entire database, or may be any subsidiary entity set. For example, functions exist to request the number of mesh entities of a given type or topology; the types and topologies are defined as

[3]Note that these descriptions do not include detailed syntax, which can be found in the interface user guide[**?**, **?**].

Table 1: Functions for Global Queries

Function	Description
iMesh_newMesh	Creates a new, empty mesh instance
iMesh_dtor	Destroys a mesh instance
iMesh_load	Loads mesh data from file into entity set
iMesh_save	Saves data from entity set to file
iMesh_getRootSet	Returns handle for the root set
iMesh_getGeometricDim	Returns geometric dimension of mesh
iMesh_getDfltStorage	Tells whether implementation prefers blocked or interleaved coordinate data
iMesh_getAdjTable	Returns table indicating availability and cost of entity adjacency data
iMesh_areEHValid	Returns true if EH remain unchanged since last user-requested status reset
iMesh_getNumOfType	Returns number of entities of type in ES
iMesh_getNumOfTopo	Returns number of entities of topo in ES
iMesh_getAllVtxCoords	Returns coords of all vertices in the set and all vertices on the closure of higher-dimensional entities in the set; storage order can be user-specified
iMesh_getEntities	Returns all entities in ES of the given type and topology
iMesh_getAdjEntities	For all entities of given type and topology in ES, return adjacent entities of adj_type
iMesh_getAllVtxCoords	For all vertices, return coords; storage order can be user-specified.
iMesh_getVtxArrCoords	For all input vertex handles, return coords; storage order can be user-specified.
iMesh_getVtxCoordIndex	For all entities of given type and topology, find adjacent entities of adj_Type, and return the coordinate indices for their vertices. Vertex ordering matches that in getAllVtxCoords.

Table 2: Functions for Single Entity Queries

Function	Description
iMesh_initEntIter	Create an iterator to traverse entities of type and topo in ES; return true if any entities exist
iMesh_getNextEntIter	Return true and a handle to next entity if there is one; false otherwise
iMesh_resetEntIter	Reset iterator to restart traverse from the first entity
iMesh_endEntIter	Destroy iterator
iMesh_getType	Return type of entity
iMesh_getTopo	Return topology of entity
iMesh_getVtxCoord	Return coordinates of a vertex
iMesh_getEntAdj	Return entities of given type adjacent to EH
iMesh_getEntArrAdj	Return entities of given type adjacent to entities of a second type adjacent to EH

enumerations. Applications can request handles for all entities of a given type or topology or handles for entities of a given type adjacent to all entities of a given type or topology. Also, vertex coordinates are available in either blocked or interleaved order. Coordinate requests can be made for all vertices or for the vertex handles returned by an adjacency call. Finally, indices into the global vertex coordinate array can be obtained for both entity and adjacent entity requests.

4.2 Entity- and Array-Based Query

The global queries described in the previous section are used to retrieve information about all entities in an entity set. While this is certainly a practical alternative for some types of problems and for small problem size, larger problems or situations involving mesh modification require access to single entities or to blocks of entities. The iMesh interface supports traversal and query functions for single entities and for blocks of entities; the query functions supported are entity type and topology, vertex coordinates, and entity adjacencies. Tables 2 and 3 summarize these functions.

4.3 Mesh Modification

The iMesh interface supports mesh modification by providing a minimal set of operators for low-level modification; both single entity (see Table 4) and block versions (see Table 5)

Table 3: Functions for Block Entity Queries

Function	Description
iMesh_initEntArrIter	Create a block iterator to traverse entities of type and topo in ES
iMesh_getNextEntArrIter	Return true and a block of handles if there are any; false otherwise
iMesh_resetEntArrIter	Reset block iterator to restart traverse from the first entity
iMesh_endEntArrIter	Destroy block iterator
iMesh_getEntArrType	Return type of each entity
iMesh_getEntArrTopo	Return topology of each entity
iMesh_getEntArrAdj	Return entities of type adjacent to each EH
iMesh_getEntArr2ndAdj	Return entities of given type adjacent to entities of a second type adjacent to each EH

Table 4: Functions for Single Entity Mesh Modification

Function	Description
iMesh_createVtx	Create vertex at given location
iMesh_setVtxCoords	Changes coordinates of existing vertex
iMesh_createEnt	Create entity of given topology from lower-dimensional entities; return entity handle and creation status
iMesh_deleteEnt	Delete EH from the mesh

of these operators are provided. High-level functionality, including mesh generation, quality assessment, and validity checking, can in principle be built from these operators, although in practice such functionality is more likely to be provided using intermediate-level services that perform complete unit operations, including vertex insertion and deletion with topology updates, edge and face swapping, and smoothing.

Geometry modification is achieved through functions that change vertex locations. Vertex locations are set at creation, and can be changed as required, for instance, by mesh smoothing or other node movement algorithms.

Topology modification is achieved through the creation and deletion of mesh entities. Creation of higher-dimensional entities requires specification, in canonical order, of an appropriate collection of lower-dimensional entities. For instance, a tetrahedron can be created using four vertices, six edges or four faces, but not from combinations of these.

Table 5: Functions for Block Mesh Modification

Function	Description
iMesh_createVtxArr	Create vertices at given location
iMesh_setVtxArrCoords	Changes coordinates of existing vertices
iMesh_createEntArr	Create entities of given topology from lower-dimensional entities; return entity handle and status
iMesh_deleteEntArr	Delete each EH from the mesh

Upon creation, adjacency information properly connecting the new entity to its components is set up by the implementation. Some implementations may allow the creation of duplicate entities (for example, two edges connecting the same two vertices), while others will respond to such a creation request by returning a copy of the already-existing entity.

Deletion of existing entities must always be done from highest to lowest dimension, because the iMesh interface forbids the deletion of an entity with existing upward adjacencies (for instance, an edge that is still in use by one or more faces or regions).

4.4 Entity Shape

Information about the shape of mesh entities is essential for support of high order accurate solution. Complicating matters, representation of curved mesh entities can be formulated in more than one way, including interpolation, approximation, analytic forms, and CAD data. In each of these formulations, however, point-wise geometric information is used to build up the required higher-order shapes of mesh entities. For example, Figure 3 shows the Lagrange interpolating and Bezier approximating shapes for mesh entities with constant or variable orders with a set of nodes used to represent the higher-order shape for mesh edges and faces. iMesh support for curved mesh entities focuses on specifying which form of geometric approximation is in use — so that an application capable of handling multiple types can distinguish between them — and the locations of the control points. Mesh shape functionality is designed to make common usage — notably equal-order Lagrange finite elements — easy, while still allowing less common, more complicated usage — such as *p*-refinement, or spectral elements, for instance. As such, global functions exist for initializing mesh entity shapes across the entire mesh, including not only creation of high-order nodes but initialization of their locations. At a more fine-grained level, nodes can be created as ordinary vertices and associated with higher-dimensional entities either entity-by-entity or node-by-node. For equal order entities, creation of and access to high order nodes for a mesh entity and its closure (for example, for the interior of a hexahedron and its bounding faces and edges) can be handled in a single call. Mixed-order elements

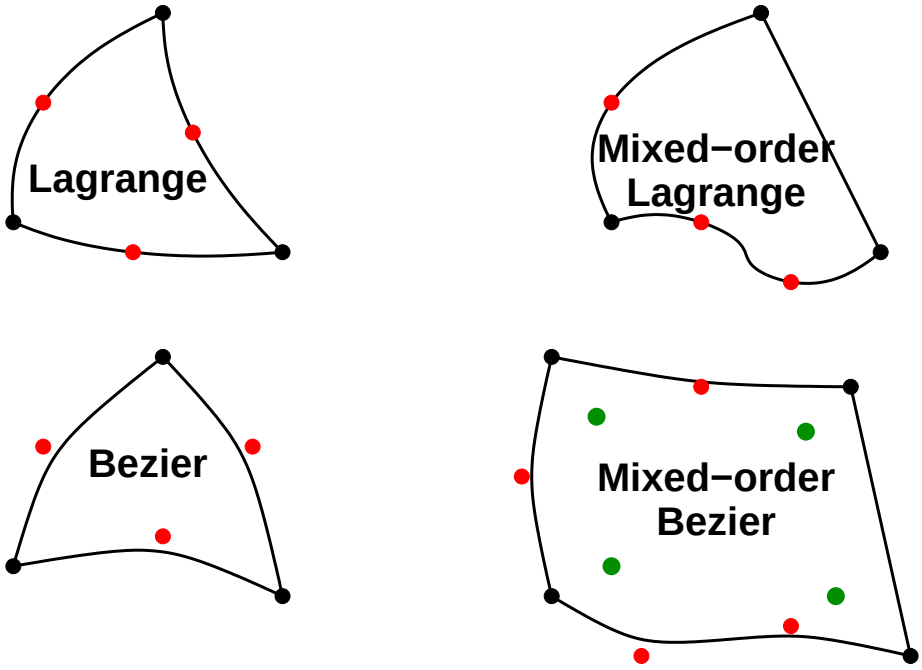


Figure 3: Examples of high-order, curved mesh entities

require a lower-level approach from the application, but we expect that writers of *p*-refined finite-element solvers will have the expertise for this. Finally, adjacency information for high-order nodes — such as the identities of all hexahedra incident on a mid-edge node — is accessed by first finding the mesh entity that a node is associated with, and then finding adjacencies for that entity.

4.5 Entity Sets

Entity set functionality in the iMesh interface is divided into three parts: basic set functionality, hierarchical set relations, and set boolean operations.

Basic set functionality, summarized in Table 7, includes creating and destroying entity sets; adding and removing entities and sets; and several entity set specific query functions. [4] Entity sets can be either ordered and non-unique, or unordered and unique; an ordered set guarantees that query results (including traversal) will always be given in the order in which entities were added to the set. The ordered/unordered status of an entity set must be

[4]Note that the global mesh query functions (Section **??**.4.1) and traversal functions (Section **??**.4.2) defined above can be used with the root set or any other entity set as their first argument.

Table 6: Functions for High-Order Entity Shape

Function	Description
iMesh_hasMeshShapes	Determine with the mesh contains high order shapes of given shape type
iMesh_createMeshShapes	Create higher order shapes with the specified shape type and order for the mesh
iMesh_hasEntShape	Determine with an entity has high order shapes of given shape type
iMesh_getEntShapeOrder	Get the order of the higher order mesh entity shape
iMesh_createEntShapes	Create high order shapes for a single entity
iMesh_deleteEntShapes	Delete high order nodes for an entity
iMesh_getEntShapes	Return high order nodes for an entity
iMesh_setVtxParam	Set parametric coordinates of high order node
iMesh_getVtxParam	Get parametric coordinates of high order node
iMesh_setNodeToEnt	Associate a high order node with a mesh entity
iMesh_getEntOfNode	Return the mesh entity on which a high order node lies
iMesh_hasEntArrShape	Determine whether an array of entities have high order shapes of given shape type
iMesh_getEntArrShapeOrder	Get the order of the high order mesh entity shape for multiple entities
iMesh_createEntArrShapes	Create high order shapes for multiple entities
iMesh_deleteEntArrShapes	Delete high order nodes for entities
iMesh_getEntArrShapes	Return high order nodes for entities
iMesh_setVtxArrParam	Set parametric coordinates of high order nodes
iMesh_getVtxArrParam	Get parametric coordinates of high order nodes
iMesh_setNodeArrToEnt	Associate high order nodes with mesh entities
iMesh_getEntArrOfNode	Return the mesh entities on which high order nodes lie

Table 7: Functions for Basic Entity Set Functionality

Function	Description
iMesh_createEntSet	Creates a new entity set (ordered and non-unique if isList is true)
iMesh_destroyEntSet	Destroys existing entity set
iMesh_isList	Return true if the set is ordered and non-unique
iMesh_getNumEntSets	Returns number of entity sets contained in SH
iMesh_getEntSets	Returns entity sets contained in SH
iMesh_addEntSet	Adds entity set SH1 as a member of SH2
iMesh_rmvEntSet	Removes entity set SH1 as a member of SH2
iMesh_isEntSetContained	Returns true if SH2 is a member of SH1
iMesh_addEntToSet	Add entity EH to set SH
iMesh_rmvEntFromSet	Remove entity EH from set SH
iMesh_addEntArrToSet	Add array of entities to set SH
iMesh_rmvEntArrFromSet	Remove array of entities from set SH
iMesh_isEntContained	Returns true if EH is a member of SH

specified when the set is created and can be queried.

Entity sets are created empty. Entities can be added to or removed from the set individually or in blocks; for ordered sets, the last of a number of duplicate entries will be the first to be deleted. Also, entity sets can be added to or removed from each other; note that, because all sets are automatically contained in the root set from creation, calls that would add or remove a set from the root set are not permitted. An entity set can also be queried to determine the number and handles of sets that it contains, and to determine whether a given entity or set belongs to that set.

Hierarchical relationships between entity sets are intended to describe, for example, multilevel meshes and mesh refinement hierarchies. The directional relationships implied here are labeled as parent-child relationships in the iMesh interface. Functions are provided to add, remove, count, and identify parents and children and to determine if one set is a child of another; see Table 8.

Set boolean operations — intersection, union, and subtraction — are also defined by the iMesh interface; these functions are summarized in Table 9. The definitions are intended to be compatible with their C++ standard template library (STL) counterparts, both for semantic clarity and so that STL algorithms can be used by implementations where appropriate. All set boolean operations apply not only to *entity* members of the set, but also to *set* members. Note that set hierarchical relationships are not included: the set result-

Table 8: Functions for Entity Set Relationships

Function	Description
iMesh_addPrntChld	Create a parent (SH1) to child (SH2) relationship
iMesh_rmvPrntChld	Remove a parent (SH1) to child (SH2) relationship
iMesh_isChildOf	Return true if SH2 is a child of SH1
iMesh_getNumChld	Return number of children of SH
iMesh_getChldn	Return children of SH
iMesh_getNumPrnt	Return number of parents of SH
iMesh_getPrnts	Return parents of SH

Table 9: Functions for Entity Set Boolean Operations

Function	Description
iMesh_subtract	Return set difference SH1-SH2 in SH
iMesh_intersect	Return set intersection of SH1 and SH2 in SH
iMesh_unite	Return set union of SH1 and SH2 in SH

ing from a set boolean operation on sets with hierarchical relationships will *not* have any hierarchical relationships defined for it, regardless of the input data. For instance, if one were to take the intersection of two directionally-coarsened meshes (stored as sets) with the same parent mesh (also a set) in a multigrid hierarchy, there is no reason to expect that the resulting set will necessarily be placed in the multigrid hierarchy at all. On the other hand, if both of those directionally-coarsened meshes contain a set of boundary faces, then their intersection will contain that set as well.

While set boolean operations are completely unambiguous for unordered entity sets, ordered sets make things more complicated. For operations in which one set is ordered and one unordered, the result set is unordered; its contents are the same as if an unordered set were created with the (unique) contents of the ordered set and the operation were then performed. In the case of two ordered sets, the iMesh specification tries to follow the spirit of the STL definition, with complications related to the possibility of multiple copies of a given entity handle in each set. We recognize that these rules are somewhat arbitrary, but have been unable to find a more systematic way of defining these operations for ordered sets. In the following discussion, assume that a given entity handle appears m times in the first set and n times in the second set.

- For intersection of two ordered sets, the output set will contain the $\min(m,n)$ copies of the entity handle. These will appear in the same order as in the first input set, with the first copies of the handle surviving. For example, intersection of the two

Table 10: Basic Tag Functions

Name	Description
iMesh_createTag	Creates a new tag of the given type and number of values
iMesh_destroyTag	Destroys the tag if no entity is using it or if force is true
iMesh_getTagName	Returns tag ID string
iMesh_getTagSizeValues	Returns tag size in number of values
iMesh_getTagSizeBytes	Returns tag size in number of bytes
iMesh_getTagHandle	Return tag with given ID string, if it exists
iMesh_getTagType	Return data type of this tag
iMesh_getAllTags	Return handles of all tags associated with entity EH
iMesh_getAllEntSetTags	Return handles of all tags associated with entity set SH

sets $A = \{abacdbca\}$ and $B = \{dadbac\}$ will result in $A \bigcap B = \{abacd\}$.

- Union of two ordered sets is easy: the output set is a concatenation of the input sets: $A \bigcup B = \{abacdbcadadbac\}$.

- Subtraction of two ordered sets results in a set containing $\min(m-n, 0)$ copies of an entity handle. These will appear in the same order as in the first input set, with the first copies of the handle surviving. For example, $A - B = \{abc\}$.

Regardless of whether the entity members of an entity set are ordered or unordered, the set members are always unordered and unique, with correspondingly simple semantics for boolean operations.

4.6 Tags

Tags are used to associate application-dependent data with a mesh, entity, or entity set. Basic tag functionality defined in the iMesh interface is summarized in Table 10, while functionality for setting, getting, and removing tag data is summarized in Table 11.

When creating a tag, the application must provide its data type and size, as well as a unique name. For generic tag data, the tag size specifies how many bytes of data to store; for other cases, the size tells how many values of that data type will be stored. The implementation is expected to manage the memory needed to store tag data. The name

Table 11: Setting, Getting, and Removing Tag Data

Function	Description
iMesh_setData	The value in tag TH for entity EH is set to the first tagValSize bytes of the array<char> tagVal
iMesh_setArrData	The value in tag TH for entities in EHarray[i] is set using data in the array<char> tagValArray and the tag size
iMesh_setEntSetData	The value in tag TH for entity set SH is set to the first tagValSize bytes of the array<char> tagVal
iMesh_set[Int,Dbl,EH]Data	The value in tag TH for entity EH is set to the int, double, or entity handle in tagVal; array and entity set versions also exist.
iMesh_getData	Return the value of tag TH for entity EH
iMesh_getArrData	Retrieve the value of tag TH for all entities in EH array, with data returned as an array of tagVal's
iMesh_getEntSetData	Return the value of tag TH for entity EH
iMesh_get[Int,Dbl,EH]Data	Return the value of tag TH for entity EH; array and entity set versions also exist.
iMesh_rmvTag	Remove tag TH from entity EH
iMesh_rmvArrTag	Remove tag TH from all entities in EH array
iMesh_rmvEntSetTag	Remove tag TH from entity set SH

string and data size can be retrieved based on the tag's handle, and the tag handle can be found from its name. Also, all tags associated with a particular entity can be retrieved; this can be particularly useful in saving or copying a mesh.

Initially, a tag is not associated with any entity or entity set, and no tag values exist; association is made explicitly by setting data for a tag-entity pair. Tag data can be set for single entities, arrays of entities (each with its own value), or for entity sets. In each of these cases, separate functions exist for setting generic tag data and type-specific data. Analogous data retrieval functions exist for each of these cases.

When an entity or set no longer needs to be associated with a tag — for instance, a vertex was tagged for smoothing and the smoothing operation for that vertex is complete — the tag can be removed from that entity without affecting other entities associated with the tag. When a tag is no longer needed at all — for instance, when all vertices have been smoothed — the tag can be destroyed through one of two variant mechanisms. First, an application can remove this tag from all tagged entities, and then request destruction of the

Table 12: Error Handling Functionality

Name	Description
iMesh_getDescription	Retrieves error description

tag. Simpler for the application is forced destruction, in which the tag is destroyed even though the tag is still associated with mesh entities, and all tag values and associations are deleted. Some implementations may not support forced destruction.

4.7 Error Handling

Like any API, the iMesh interface is vulnerable to errors, either through incorrect input or through internal failure within an implementation. For instance, it is an error for an application to request entities with conflicting types and topologies. Also, an error in the implementation occurs when memory for a new object cannot be allocated. The iMesh interface defines a number of standard error conditions which could occur in iMesh functions, either because of illegal input or internal implementation errors; each of these error conditions has an accompanying description, which can be retrieved by calling iMesh_getDescription.

4.8 Fortran Compatibility

For compatibility with the Fortran convention that functions returning values do not modify their arguments, no iMesh function returns a value. That is, all iMesh functions are C void functions or Fortran subroutines. Also, string arguments in the C API have an accompanying argument giving their length; these string length arguments are added at the end of the argument list in the order the strings appear. Finally, the iMesh API requires the use of a Fortran compiler that supports the common pass-by-value extension.

5 Usage Examples

This section will replace the current back end of the paper.

Examples I have in mind to discuss in some detail, both as usage examples and to discuss efficiency issues, etc:

- Tim's basic example.
- Swapping service. Memory mention.

- Mesquite?
- Lori's FE example
- Zoltan
- Coarse-grained adaptivity?

Then I'd like to be able to say a few words about some of the more complex examples that use iMesh, including things like SLAC / DDriv (Tim) and groundwater (Harold). Once we've got a consensus on which examples and roughly the level of detail we're after, I'll assign homework.

6 Discussion and Conclusions

Status: services, implementations
Directions

References

[1] S. Balay, K. Buschelman, D. Gropp, W.D. Kaushik, M. Knepley, B.F. McInnes, L.C. Smith, and H. Zhang. PETSc home page. `http://www.mcs.anl.gov/petsc`, 2004.

[2] S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In A.M. Bruaset E. Arge and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[3] Kyle Chand, Lori Freitag Diachin, Brian Fix, Carl Ollivier-Gooch, E. Seegyoung Seol, Mark S. Shephard, and Timothy Tautges. Toward interoperable mesh, geometry and field components for PDE simulation development. *Submitted to Engineering with Computers*, 2005.

[4] Kyle Chand, Brian Fix, Tamara Dahlgren, Lori Freitag Diachin, Xiaolin Li, Carl Ollivier-Gooch, E. Seegyoung Seol, Mark S. Shephard, Tim Tautges, and Harold Trease. The TSTTB Interface. `https://svn.scorec.rpi.edu/svn/TSTT/Documentation/TSTTB_userguide.pdf`, November 2005.

[5] Kyle Chand, Brian Fix, Tamara Dahlgren, Lori Freitag Diachin, Xiaolin Li, Carl Ollivier-Gooch, E. Seegyoung Seol, Mark S. Shephard, Tim Tautges, and Harold Trease. The TSTTM Interface. `https://svn.scorec.rpi.edu/svn/TSTT/Documentation/TSTTM_userguide.pdf`, November 2005.

[6] Eispack webpage. `http://www.netlib.org/eispack/`, 2004.

[7] Lapack webpage. `http://www.netlib.org/lapack/`, 2004.

[8] Linpack webpage. `http://www.netlib.org/linpack/`, 2004.

[9] Mark S. Shephard. Meshing environment for geometry-based analysis. *Int. J. Numer. Meth. Engng.*, 47:169–190, 2000.

[10] J. Steinbrenner, T. Michal, and J. Abelanet. An industry specification for mesh generation software. In *Proceedings of the 17th AIAA Computational Fluid Dynamics Conference*. American Institute for Aeronautics and Astronautics, 2005.

[11] Unstructured Grid Consortium Standards Document. `http://www.aiaa.org/tc/mvce/ugc/ugcstandv1.pdf`, 2002.

[12] The Unstructured Grid Consortium. `http://www.aiaa.org/tc/mvce/ugc/`, 2005.

7 iMesh Interface Syntax

Also, each array argument has an accompanying integer telling how many entries in the array are in use; output arrays also have an integer argument specifying their total allocated size. In addition to the arguments listed, each function also has a mesh instance as its first argument (analogous to the hidden `this` argument in C++ member functions) and returns an integer status value in an argument.

8 Specific Function Example

An example of a specific function, as written in the C API, as called from Fortran, as specified by SIDL, and as called from at least one language using Babel.