SCOREC Mesh Database

Version 2.4

Users Guide

Mark W. Beall

Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, NY 12180

August 2, 1996

1.0 Introduction

This section is a general introduction to the SCOREC Mesh Database. Some of the basic concepts behind the mesh database are described. In addition some general instructions on linking to and using the mesh database are given as well as some debugging tips.

1.1 Current Status

This document describes version 2.4 of the SCOREC Mesh Database. The software described here has been used in several major programs and many minor ones without problems. At this point most of the bugs should have been worked out (although certainly having said that, new ones will appear). This software is written in ANSI C and thus should be portable to any system that has an appropriate compiler, it has been used with the following compilers and operating systems:

SunOS 4.1.3 (acc and gcc)

SunOS 5.4, 5.5 (cc)

IRIX 5.3, 5.4 (cc)

AIX 3.2, 4.1 (cc and gcc)

1.2 Overview

The SCOREC Mesh Database is designed to allow easy access to and modification of data that describes the discretization of a geometric domain. Specifically, the database has been designed for use in the context of finite element meshes, but it certainly may have applicability outside that one area.

The mesh database is implemented using an object-oriented methodology. There is a set of objects defined that represent the entities in the database. Each of the objects has a set of operators that act to retrieve/store/modify data in the database. There is no access to the actual internal data structures in the database. The objects that exist in the mesh database are as follows:

pMesh - a model in which the entities in are classified with respect to a geometric model.

pGModel - a topological representation of a geometric model.

pRegion - a three dimensional entity bounded by faces.

pFace - a two dimensional entity bounded by edges.

pEdge - a one dimensional entity bounded by vertices.

pVertex - a zero dimensional entity.

pPoint - a position in x, y, z space.

pNode - something that degrees of freedom are associated with.

pPList - variable length list

There are two other objects that are base classes of the above objects, when an operator takes one of these objects it may take an object of any of its derived types, when an operator returns one of these objects it may be an object of any of its derived types:

pModel - either a pMesh or a pGModel

pEntity - a pRegion, pFace, pEdge, pVertex or pNode.

Each operator takes as its first argument the object to be operated on.

The mesh database can be operated in different modes, which have different internal data representations and varying degrees of storage and speed, depending on the users needs. There is a mode that determines whether the database is static or dynamic. In the static mode the user may only retrieve data from the database, the data stored in the database may not be altered. The dynamic mode, which allows modification of the database, uses dynamic data structures internally which take up additional storage. The other mode supported determines the type of topological data representation that the database uses. A modification of the Radial Edge Data Structure, called the Reduced Mixed-Mesh Data Structure may be used. This topological representation includes "uses" in the representation of the mesh to allow for the unambiguous representation of non-manifold topologies. The other topological representation does not use entity uses in its representation, thus saving storage.

1.3 Debugging Tips

Each of routines of the mesh database does run-time type checking on some of it's arguments. The only arguments that are type checked are those that are of the following types: pRegion, pFace, pEdge and pVertex. If a wrong argument is found the message "typecheck Error" will be printed on the standard output. In order to debug this type of problem set a break point in the routine "typecheckError". This routine is called whenever a type checking error is found. By then looking at the function call stack, you can determine where the offending function call was made from.

1.4 History

1.4.1 Version 2.4

- Creation of mesh entities can now be done with one operator (see MM_create* operators). Old operators used for this purpose have been depreciated and will be removed in future versions. See "Depreciated Operators" on page 30.
- Considerable improvements in both storage space and speed.
- Distinction between static and dynamic meshes and entities is almost completely eliminated. Future versions will support full functionality (equivalent of dynamic) for all meshes.

1.4.2 Version 2.3

- pNodes now derived type of pEntity.
- Minor bug fixes.

1.4.3 Version 2.2

- Improved performance of list objects.
- Other minor performance improvements.
- Minor bug fixes.

1.4.4 Version 2.1

- Preliminary Octant and octree operators added.
- Several bug fixes in dynamic mesh data structures.

1.4.5 Version 2.0

- The first full version of the mesh database.
- Support added for dynamic mesh data structures.

1.5 Getting Help/Reporting Problems

This document is supposed to give you all the information that you need to do anything that you could possibly want to do with the mesh database. However, this is more or less impossible since I could never foresee everything anyone would want to try to do. If you have problem using the SCOREC mesh database or suggestions for enhancing it, please send me mail at the address below.

If you believe that you have found a bug in the mesh database, I would really like to hear about it. In order to assist in isolating the bug I ask that you try to demonstrate it with a short piece of code (the shorter, the better). Send bug reports to the address below.

All correspondence regarding the SCOREC Mesh Database should be send to:

by email (preferred):

mbeall@scorec.rpi.edu

or:

Mark Beall
Scientific Computation Research Center
CII 7011
Rensselaer Polytechnic Institute
Troy, NY 12180

2.0 Database Query Operators

This section describes the operators for querying information from the mesh database. Operators for modifying the data stored in the mesh database may be found in Section 3.0 on page 20.

The naming convention for operators is as follows:

- **operator name**
- input parameters
- *output parameters*
- *input/output parameters*

For each operator a description of the operator is given and then a description of any required arguments in addition to the object (the first argument) is given.

Many of the operators return a list that is of type pPList. The operators that operate on these lists can be found in Section 4.1 on page 23.

Note: The operator PList_delete must be called for eachlists returned by any of the operators or memory leaks and poor performance will result.

2.0.1 Using topological adjacency operators with geometric model entities

Although the database described here is a mesh database, it can also be used to provide some information about the topology of the geometric model that the emsh was created from. (Eventually this capability will be provided by a seperate library that interfaces directly with the appropriate modeler). There are some limitations as to what topological adjacency operators may be called for entites from the geometric model. Basically, the operators that return multi-level adjacencies should not be called. The following topological adjacency operators are save to call for geometric model entities: R_face, F_edge, F_region, E_vertex, E_face, V_edge. Any of the operators that return information other than adjacency information (e.g. R_numFaces) are safe to call.

2.0.2 Calling the operators from Fortran

Each of the operators may be called from Fortran, the name is the same as the C routine, but if the C routine returns a value (has a return type that is other than void), that value will be returned in an extra parameter in the function call. In addition all of the parameter of the types p* (e.g. pEntity, pFace, etc.) are integers. For example,

int EN_id(pEntity entity)

would become

EN_id(INTEGER entity, INTERGER *id*)

2.1 Database Operators

MD_init void MD_init(void)

Initializes the mesh database. This operator *must* be called before any other operators in the mesh database may be called.

MD_exit void MD_exit(void)

This operator should be called after all work is done with the mesh database and all object have been deleted. No mesh operators may be called after this operator has been called. MD_init may be called again after MD_exit to reinitialize the mesh database.

MD_debugListCacheOff

void MD_debugListCacheOff(void)

This operator is provided to use with debugging tools which check for memory leaks such as Purify and Sentinel. This will turn off the mesh databases caching of list objects which can make memory leaks appear to be coming from somewhere other than where they are actually occuring. This will decrease the performance of the mesh database. This operator should be called immediately after MD_init() and before any other mesh database operators.

2.2 General Entity Operators

The general entity operators, EN_*, may be called for any of the following objects: pRegion, pFace, pEdge, pVertex, pNode and pOctant. Some of the operators may not make sense for some of these entities (e.g. attaching a node to a node).

EN_type *Type* EN_type(pEntity)

Return the type of pEntity. Type is one of the following: Tregion, Tface, Tedge, Tvertex, Toctant, Tnode. These symbols are defined in the header file for the mesh operators. The definitions of Type are such that for a region, face edge and vertex the value corresponds to the dimension of the entity.

Fortran note: For Fortran the types are defined as follows:

0=vertex, 1=edge, 2=face, 3=region, 5=octant, 6=node

EN_tag

int EN_tag(pGEntity entity)

Returns the numeric tag associated with pGEntity. This tag is a number that the modeler that created the model uses to identity the entity. This operators only makes sense to call if entity is from a geometric model (not a mesh).

EN_setID

void EN_setID(pEntity entity, int id)

Set a numerical id for the given entity. This id is returned by the operator EN_id. This id may be changed by the mesh database if the operator M_writeSMS is called.

EN_id

int EN_id(pEntity entity)

Returns the numberical id for the given entity. This id may be set using the operator EN_setID.

2.2.1 Node Operators

The following two operators are used to attach nodes to entities and to retrieve the nodes once attached. Nodes attached to an entity are identified by the entity and a number. The convention is that nodes with numbers from zero to the number of points on an entity are associated with the points attached to the entity. Nodes with numbers less than zero are associated with the entity itself. Note that a node itself does not have coordinates, nodes that are associated with a point on an entity have the location of the point on the entity with which they are associated. Operators that act on nodes are given in Section 2.8 on page 15.

EN_setNode

void EN_setNode(pEntity entity, int n, pNode node)

Attaches node to point n on entity. The node that is being attached must have been previously created using the operator N_new. A single node may be attached to each point on an entity (i.e. if there are p points you may attach p different nodes, one at each point). In addition, multiple nodes may be attached to the entity itself.

EN_node

pNode EN_node(pEntity entity, int n)

Retrieves the node attached to point n on entity. Returns null if no node is attached to that point. If n<0 then retrieve the node associated with the entity itself.

2.2.2 Data Operators

The following operators are used to attach user defined data to any entity in a model. Two types of data may be attached to entities, either a pointer or a integer. The data is identified by a tag that is a four character string. When passing the tag to the operators the string must be at least four characters (only the first four of which are significant). For tags less than four characters the rest must be padded with blanks (note: the position of the blanks is significant " tag" and "tag " are different tags). Currently there is no provision for saving user defined data if the mesh is written to a file.

EN_attachDataP void EN_attachDataP(pEntity entity, char *tag, void *data)

Attach the pointer data to entity with the id tag. See the above paragraph for a description of what makes a valid tag. tag must be unique on the entity, if there is already data with the tag tag on the entity the behavior of the operator is undefined. To modify existing data use the operator EN_modifyDataP.

EN_dataP *void* * EN_dataP(pEntity entity, char *tag)

Retrieve the data with id tag from entity, if the tag does not exist on the entity a null pointer is returned.

EN_modifyDataP *int* EN_modifyDataP(pEntity entity, char *tag, void *data)

Replace the existing data with id tag on entity with data. The data pointed to by the original pointer is not deleted, so unless a reference to this data is kept elsewhere this may cause memory leaks. The return value will be zero if the tag does not exist on the entity.

EN_attachDataI void EN_attachDataI(pEntity entity, char *tag, int data)

Attach the integer data to entity with the id tag. See note after description of EN_dataI for a restriction on using this operator. tag must be unique on the entity, if there is already data with the tag tag on the entity the behavior of the operator is undefined. To modify existing data use the operator EN_modifyDataI.

EN_dataI *int* EN_dataI(pEntity entity, char *tag)

Retrieve the data with id tag from entity, if the tag does not exist on the entity a zero is returned.

Note: that since a zero is returned if the tag does not exist it is not possible to use this operator to test for the existence of data with the given tag unless zero is not a valid value for the data.

EN_modifyDataI *int* EN_modifyDataI(pEntity entity, char *tag, int data)

Replace the existing data with id tag on entity with data. The return value will be zero if the tag does not exist on the entity.

EN_removeData void EN_removeData(pEntity entity, char *tag)

Remove the data with id tag from entity. In the case of pointer data, the data being pointed to is not disposed of (i.e. free() is not called). The user must free the memory that the pointer points to to avoid memory leaks.

2.3 Region Operators

R_whatIn *pGEntity* R_whatIn(pRegion region)

Returns the entity on the geometric model that region is classified on.

R_numFaces *int* R_numFaces(pRegion region)

Returns the number of faces that enclose region.

R_face *pFace* R_face(pRegion Region, int n)

Returns face n on Region

n: Index of face on region to return. The first face is n=0.

R_faceDir *int* R_faceDir(pRegion Region, int n)

Returns the direction of face n on Region. Returns 1 if the face is being used such that the normal of the face points outside the region, returns 0 otherwise.

n: Index of face on region to return direction for. The first face is n=0.

R_faces *pList* R_faces(pRegion region)

Returns a list of faces on region. The faces are returned in *list*.. The list returned by this operator must be deleted (using the operator PList_delete) to avoid memory leaks.

R_edges *pList* R_edges(pRegion region)

Returns a list of edge on region. The edges are returned in the order shown. The order for other shaped regions is undefined. The vertex

ordering in Figure 2 and edge ordering in Figure 1 are correct relative to one another (i.e. edge 3 is between vertex 0 and vertex 3).

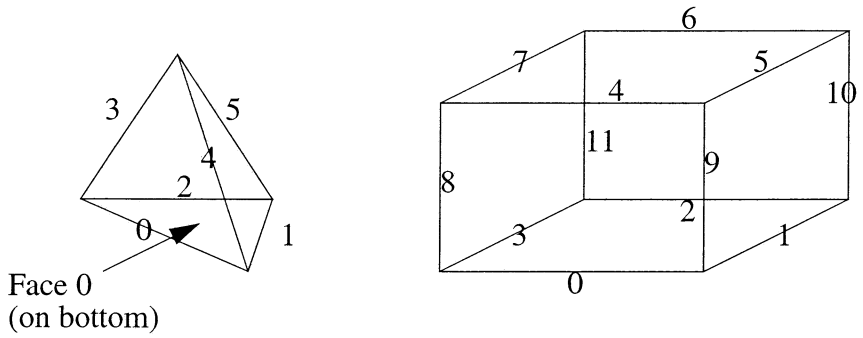

FIGURE 1. Edge Order on a Region

R_vertices *pList* R_verticies(pRegion region)

Returns a list of vertices that are on region. The vertices are returned in the order shown. The order for tother shaped regions is undefined.

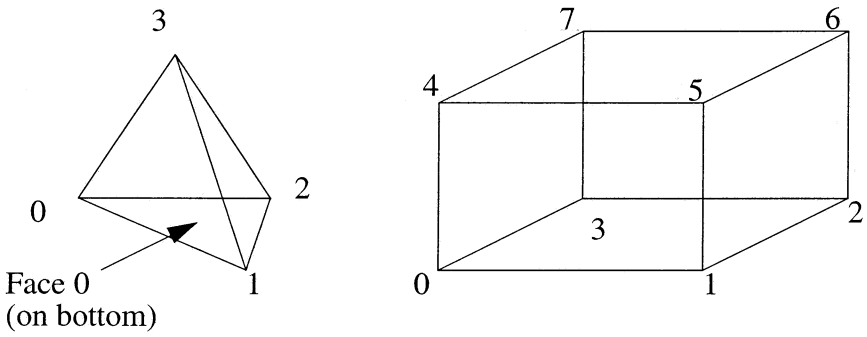

FIGURE 2. Vertex Order on a Region

R_inClosure *int* R_inClosure(pRegion region, pEntity entity)

Checks if entity is inthe closure of region. entity can be a pRegion, pFace, pEdge, or pVertex. Returns 1 if entity is in the closure of region return 0 if it is not.

R_dirUsingFace int R_dirUsingFace(pRegion region, pFace face)

Returns the direction region is using face.

2.4 Face Operators

F_whatIn *pGEntity* F_whatIn(pFace face)

Returns the entity on the geometric model that face is classified on.

F_numEdges *int* F_numEdges(pFace face)

Returns the number of edges that enclose face.

F_edge *pEdge* F_edge(pFace face, int n)

Returns the n^{th} edge on face. The first edge is n=0. Edges are counter-clockwise, with respect to the normal, around face.

F_edgeDir *int* F_edgeDir(pFace face, int n)

Returns the direction in which face is using edge n. 1=using in the direction that edge n is defined in, i.e. in the direction from the first to the second vertex. The first edge is n=0.

F_region *pRegion* F_region(pFace face, int n)

Returns region n (n=0,1) on face. Returns null if no region is attached to that side of the face.

n: n=1 is the region on the side of the positve normal to the face.

F_regions *pList* F_regions(pFace face)

Returns a list containing the two regions on a face.

F_edges *pList* F_edges(pFace face, int dir, pVertex vtx)

Returns the edges on face, in the direction dir, starting with the edge that begins with vtx.

dir: If dir =1 then edges are returned in the direction that face is defined in, otherwise they are returned in the opposite direction.

vtx: If vtx is specified the edges are returned in the order such that the first edge has vtx on the end opposite the vertex shared with the second edge. If vtx is not specified (i.e. passed in as null) then the edges will be returned with the first edge being the first edge that was used to define face.

For the face shown below (the positive normal is coming out of the page), the call F_edges(F_1, 1, V_2) will return E_2, E_3, E_1, the call F_edges(F_1,0,V_1) will return E_3, E_2, E_1

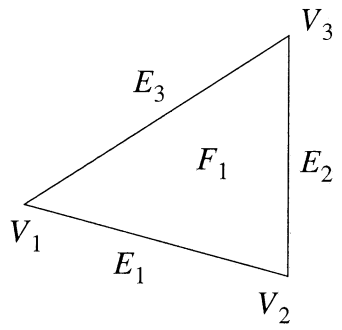

FIGURE 3. Ordering for F_edges

F_vertices *pList* F_verticies(pFace face, int dir)

Returns the vertices on face in the direction dir.

dir: If dir =1 then vertices are returned in the direction that face is defined in (counterclockwise around the normal), otherwise they are returned in the opposite direction.

F_inClosure *int* F_inClosure(pFace face, pEntity entity)

Checks if entity is in the closure of face. entity can be a pFace, pEdge or pVertex. Returns 1 if entity is in the closure of face, return 0 if it is not.

F_dirUsingEdge int F_dirUsingEdge(pFace face, pEdge edge)

Returns the direction face is using edge.

2.5 Edge Operators

E_whatIn *pGEntity* E_whatIn(pEdge edge)

Returns the entity on the geometric model that edge is classified on.

E_vertex *pVertex* E_vertex(pEdge edge, int n)

Returns the n^{th} (n=0,1) vertex on edge.

E_numFaces *int* E_numFaces(pEdge edge)

Returns the number of faces using edge.

E_face *pFace* E_face(pEdge edge, int n)

Returns the n^{th} face using edge. The first face is n=0. The faces are not returned in any particular order.

E_regions *pList* E_regions(pEdge edge)

Returns a list of the regions that edge is in the closure of.

E_inClosure *int* E_inClosure(pEdge edge, pEntity entity)

Checks if entity is inthe closure of edge. entity can be a pEdge or pVertex. Returns 1 if entity is in the closure of edge, return 0 if it is not.

E_numPoints *int* E_numPoints(pEdge edge)

Returns the number of interior points on edge.

E_point *pPoint* E_point(pEdge edge, int n)

Returns the n^{th} interior point on edge. The first point is n=0. If there is more than one point they are ordered along the edge such that the first one is closest to vertex 0 and the list one is closest to vertex 1.

E_otherFace *pFace* E_otherFace(pEdge edge, pFace face, pRegion region)

Returns the face other than face on edge connected to region. If face=NULL the operator returns any face bounding region on edge. Returns a NULL if no face satisfying the given conditions can be found.

Note: this operator only makes sense to use if the edges is part of a mesh (not a geometric model).

E_otherVertex pVertex E_otherVertex(pEdge edge, pVertex vertex)

Returns the vertex on edge that is not vertex.

E_uses *pList* E_uses(pEdge edge)

***

2.6 Vertex Operators

V_whatIn *pGEntity* V_whatIn(pVertex vertex)

Returns the entity on the geometric model that vertex is classified on.

V_numEdges *int* V_numEdges(pVertex vertex)

Return the number of edges using vertex.

V_edge *pEdge* V_edge(pVertex vertex, int n)

Return the n^{th} edge using vertex. The first edge is n=0. The edges are not returned in any particular order.

V_faces

V_regions *pList* V_regions(pVertex vertex)

Returns a list of the regions that vertex is in the closure of.

V_point *pPoint* V_point(pVertex vertex)

Return the point associated with vertex.

V_uses *pList* V_uses(pVertex vertex)

Returns a list of the vertex uses that exist for vertex. If the vertex has no uses then NULL is returned.

VU_faces

VU_edges

2.7 Point Operators

P_x *double* P_x(pPoint point)
P_y *double* P_y(pPoint point)
P_z *double* P_z(pPoint point)

Return the x, y, z coordinates of point

P_id *int* P_id(pPoint point)

Return the id of the point

P_setID void P_setID(pPoint point, int id)

Set the id of the point

P_param1 *double* P_param1(pPoint point)

Retrieves a single parameter value for point. To be used if point is on an entity classifed on a model edge.

P_param2 void P_param2(pPoint point, double *r, double *s, int *patch)

Retrieves two paramter values (and an integer patch indicator) for point. To be used if point is on an entity classified on a model face.

P_setPos void P_setPos(pPoint point, double x, double y, double z)

Set the coordinates of point to x,y,z.

P_setParam1 void P_setParam1(pPoint point, double r)

Sets a single paramter value for point. To be used if point is on a mesh entity classifed on a model edge.

P_setParam2 void P_setParam2(pPoint point, double r, double s, int patch)

Sets two parameter values (and an interger patch indicator) for point. To be used if point is on a mesh entity classified on a model face.

2.8 Node Operators

These operators are for manipulating nodes. For operators to set and retrieve nodes on mesh entities see Section 2.2.1 on page 7. Note that a node itself does not have coordinates, nodes that are associated with a point on an entity have the location of the point on the entity with which they are associated.

N_new *pNode* N_new(void)

Creates a new node.

N_num *int* N_num(pNode node)

Return the number associated with node.

N_setNum void N_setNum(pNode node, int n)

Sets the number associated with node to n.

N_delete void N_delete(pNode node)

Deletes node. This does not delete any references to the node, just the node itself.

N_whatIn

N_point

2.9 Model Operators

The term model is used to refer to a datatype that is either a topological representation of a mesh or of a geometric model. If an argument is specified to be a pModel then either a pMesh or a pGModel can be passed.

MM_new

pMesh MM_new(int type, GMODEL model)
MM_NEW(INTEGER type, GMODEL model, MESH mesh)

Creates a new mesh based on type that is classified on model.

type: pass a 0 (zero) for this argument if the mesh will not be modified, pass 1 if the mesh will be modified (this will create a dynamic data structure for the mesh and make all of the entities in the mesh dynamic, this option does require more memory to store the mesh)

model: object returned by operator GM_new(). You may pass a 0 (zero) if there is no model associated with the mesh (operators returning classification information should not be called in this case). The model must be loaded using M_load before the mesh is loaded although the model may be loaded after it is passed to MM_new.

GM_new

pGModel GM_new(int type)

Creates a new geometric model based on type.

type: pass a 0 (zero) for this argument.

M_delete

void M_delete(pModel model)

Deletes the object model and releases all memory associated with it. All entities on model are also deleted.

M_load

int M_load(pModel model, char *filename)

Loads a model from the file filename. The only currently supported file formats are the SCOREC mesh or model format which are generally indicated by a .sms or .smd extension, respectively. The file extension is part of filename. The return value is zero if the specified file could not be opened, non-zero otherwise.

M_writeSMS

void M_writeSMS(pMesh mesh, char *filename, char *program)

Writes out a file in the SCOREC mesh format for the given mesh. The filename is the full name of the file that will be written (i.e. a .sms is not

appended to the name given). program should be the name of the program that is writing out the mesh file, this name is written to the .sms file. The string program may have no space characters.

M_writeDXfile void M_writeDXfile(pMesh mesh, char *name)

***

M_nRegion *int* M_nRegion(pModel model)
M_nFace *int* M_nFace(pModel model)
M_nEdge *int* M_nEdge(pModel model)
M_nVertex *int* M_nVertex(pModel model)
M_nPoint *int* M_nPoint(pModel model)

Each of the above operators return the number of the appropriate entities that are on model.

M_nextRegion *pRegion* M_nextRegion(pModel model, void **flag)
M_nextFace *pFace* M_nextFace(pModel model, void **flag)
M_nextEdge *pEdge* M_nextEdge(pModel model, void **flag)
M_nextVertex *pVertex* M_nextVertex(pModel model, void **flag)
M_nextPoint *pPoint* M_nextPoint(pModel model, void **flag)

Each of the above operators when called repeatedly will return each of the appropriate type of entity in the model. To initialize the operators to the first entity in the mesh flag must be set to point to a null value. When the operator is called for the entity after the last one in the model it will return a null value. An example of using these operators is below.

```
pRegion region;
void *temp = 0;
while ( region = M_nextRegion(mesh, &temp) ) {
... do something ...
}
```

Important: the operators below should be avoided if at all possible (the M_next* operators should be used instead), it can be very inefficient, especially in the case of dynamic models.

M_region *pRegion* M_region(pModel model, int n)
M_face *pFace* M_face(pModel model, int n)
M_edge *pEdge* M_edge(pModel model, int n)
M_vertex *pVertex* M_vertex(pModel model, int n)
M_point *pPoint* M_point(pModel model, int n)

Each of the above operators return the n^{th} entity of the given type on model. Note: the first entity is n=0. These operators should generally not be used and the M_next* operators should be used in their place.

n: index of entity on model.

Note: the M_next* operators should be used instead of these operators to loop through all the entities in the model. It is possible for these operators to be very inefficient, especially in the case of dynamic models.

2.10 Mesh Operators

These operators operate only on meshes. They may be used if a mesh if static or dynamic.

MM_makeNodes void MM_makeNodes(pMesh mesh, nodeType type)

Create nodes on the mesh corresponding to a specific type of node pattern (which determines which entities the nodes are created on).

type: Valid types are (numerical values in parenthesis should be passed if calling from Fortran):

MM_isoparametric (0): creates a node at each vertex and one associated with each point on any other entities.

MM_nNode *int* MM_nNode(pMesh mesh)

Returns the number of nodes on mesh.

MM_addNode void MM_addNode(pMesh mesh, pNode node)

Adds a node to the mesh. This operators just registers that the given node is a part of the mesh it does not affect any of the information about that node.

MM_nextNode *pNode* MM_nextNode(pMesh mesh, void **flag)

This operator, when called repeatedly will return each of the nodes in the mesh. To initialize the operator to the first node in the mesh flag must be set to point to a null value. An example of using this operator is below.

```
pNode node;
void *temp = 0;
while ( node = MM_nextNode(mesh, &temp) ){
... do something ...
}
```

MM_removeNode void MM_removeNode(pMesh mesh, pNode node)

Deletes the node from mesh.

3.0 Database Modification Operators

The following operators are used to modify the information in the mesh database. They may only be used if a dynamic mesh has been created.

3.1 Region Operators

R_setWhatIn void R_setWhatIn(pRegion region, pEntity model_entity)

Set the classification of region to model_entity.

3.2 Face Operators

F_setWhatIn void F_setWhatIn(pFace face, pGEntity what)

Set the classification of face to what.

F_chDir void F_chDir(pFace face)

Flips face so that its positive normal is pointing in the opposite direction. This operator updates all of the entities adjacent to face to reflect this change (the direction that the face is being used by any regions is reversed and the direction that it is using any edges is reversed).

3.3 Edge Operators

E_setWhatIn void E_setWhatIn(pEdge edge, pGEntity what)

Set the classification of edge to what.

E_setNpoint void E_setNpoint(pEdge edge, int n)

Set the number of points on edge to n. Currently this operator does nothing and there is a limit of one point per edge. However this operator should be called to assure compatibility with future versions of the mesh database that will not have the above assumption.

E_setPoint void E_setPoint(pEdge edge, pPoint pt)

Set point pt to be on edge.

3.4 Vertex Operators

V_setWhatIn void V_setWhatIn(pVertex vertex, GEntity what)

Set the classification of vertex to be on what.

V_setPoint void V_setPoint(pVertex vertex, pPoint pt)

Set the point associated with vertex to be pt. There can only be one point associated with a vertex.

3.5 Point Operators

P_new *pPoint* P_new(void)

Create a new point.

P_delete void P_delete(pPoint point)

Delete point. This operator should not be called if the point is part of a mesh (i.e. if it has been attached to a mesh using the M_addPoint operator). In this case the M_removePoint operator should be called instead so that the mesh knows that the point has been deleted.

3.6 Model Operators

The term model is used to refer to a datatype that is either a topological representation of a mesh or of a geometric model. Where an argument of type pModel is given either a pMesh or a pGModel may be passed.

M_removeRegion void M_removeRegion(pModel model, pRegion region)
M_removeFace void M_removeFace(pModel model, pFace region)
M_removeEdge void M_removeEdge(pModel model, pEdge region)
M_removeVertex void M_removeVertex(pModel model, pVertex region)
M_removePoint void M_removePoint(pModel model, pPoint region)

Deletes the appropriate entity from the model and updates the entities pointing to the deleted entity.

3.7 Mesh Operators

These operators operate only on meshes.

MM_createR *pRegion* MM_createR(pMesh mesh, int nFace, pFace *faces, int *dirs, pGEntity gent)

Returns a new region in the given mesh.

nFace: number of faces that define the region

faces: array (size=nFace) of faces bounding the region

dirs: array (size=nFace) of integers indicating the direction each face in faces is being used by the region (1=positive direction, 0 = negative direction)

gent: entity in the geometric model that this region is classified on

MM_createF

pFace MM_createF(pMesh mesh, int nEdge, pEdge *edges, int *dirs, pGEntity gent)

Returns a new face in the given mesh.

nEdge: number of edges that define the face

edges: array (size=nEdge) of edges bounding the face. The edges in this array should be given in order around the face.

dirs: array (size=nEdge) of integers indicating the direction each edge in edges is being used by the face (1=positive direction, 0 = negative direction).

gent: entity in the geometric model that this face is classified on

MM_createE

pEdge MM_createE(pMesh mesh, pVertex v1, pVertex v2, pGEntity gent)

Returns a new edge in the given mesh.

v1, v2: the two vertices bounding the edge

gent: entity in the geometric model that this edge is classified on

MM_createVP

pVertex MM_createVP(pMesh mesh, double x, double y, double z, double *param, int unused, pGEntity gent)

Creates a new vertex and point in the given mesh. Returns only the vertex.

x,y,z: spatial location of the vertex

param: array containing the parametric location of the vertex on the entity it is classified on, ent. Size of this array equals the dimension of the parametric space (1 if ent is an edge, 2 if a face). Not used (pass 0) for vertices classifed on a vertex or a region.

unused: unused parameter - pass 0.

gent: entity in the geometric model that this vertex is classified on

4.0 Utility Operators

4.1 pList Operators

A pList is a datatype that allows the manipulation of arbitrary length lists of Entities. The following operators are defined for pLists.

PList_new

pList PList_new(void)

Creates a new pList.

PList_delete

void PList_delete(pList list)

Deletes list. Does not delete the entities that are in list.

PList_append

pList PList_append(pList list, pEntity item)

Appends item to list.

PList_appUnique

pList PList_appUnique(pList list, pEntity item)

Appends item to list only if item is not already in list. This operator is less efficient than PList_append since it must search the list before adding the item.

PList_size

int PList_size(pList list)

Returns the number of items in the list.

PList_item

pEntity PList_item(pPList list, int n)

Returns the n^{th} item in list. The first item in the list is n=0.

PList_remItem

void PList_remItem(pPList list, pEntity item)

Removes the given item from the list.

PList_next

PList_clear

5.0 Octree Operators

These operators are a preliminary set of operators for accessing octant information that is related to the mesh. Although the operators exist, the data that the operators use is not currently automatically loaded into the database. If you desire to use these operators please contact me (See "Getting Help/Reporting Problems" on page 3.)

5.1 Octant Operators

An octant has been implemented as a type of pEntity. Therefore in addition to the operators listed here any of the operators listed in "General Entity Operators" on page 6 may be used.

When child octants are referred to by number, the numbering shown in the figure below is to be used.

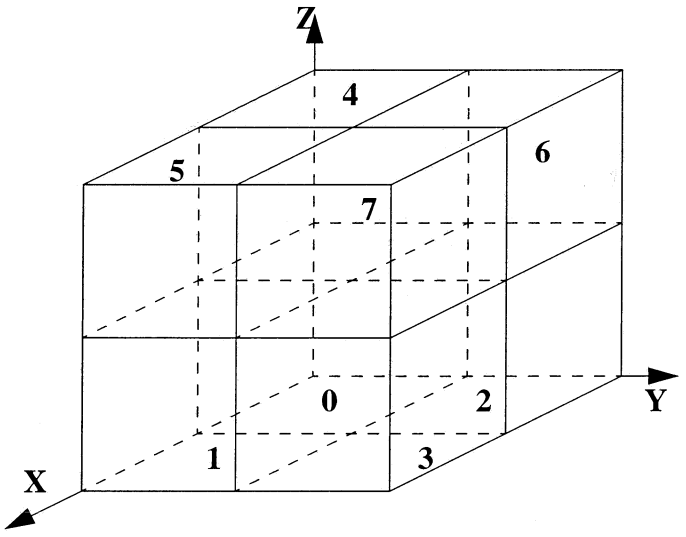

FIGURE 4. Child octant numbering

OC_new pOctant OC_new(pOctant parent, int whichChild)

Creates a new Octant. If the parent of the octant is given OC_setChild will be called for the parent octant to notify the parent that the new octant is it's child. The level of the new octant will be set to one greater than it's parent. The range for whichChild is 0 to 7. To create a root octant pass 8 for whichChild and the octree that the octant is the root of for parent.

OC_delete void OC_delete(pOctant octant)

Deletes octant. If octant is terminal removes the association of the octanta with the entities that were inside of it. If octant is non-terminal, recursively decends the tree, deleteing everything below octant by calling OC_delete on each child.

OC_parent *pOctant* OC_parent(pOctant octant)

Returns the parent octant of the given octant.

OC_child *pOctant* OC_child(pOctant octant, int n)

Returns the n^{th} child of octant. n must be between 0 and 7. Returns null if octant is a terminal octant.

OC_children int OC_children(pOctant, pOctant **children*)

Fills the array, which must be at least of size 8, with the children of the octant.

OC_setChild void OC_setChild(pOctant, pOctant child, int n)

Stores child as a the n^{th} child octant of the given octant.

OC_deleteCh void OC_deleteCh(pOctant)

Deletes the child octants of the given octant. Does nothing if the octant is a terminal octant. If the child octants are not terminal also deletes their children. The list of entities set up by previous calls to OC_setEnt will be deleted from any octants before the octant is deleted. Any data attached with EN_attachData should be removed before an octant is deleted or memory could be leaked.

OC_level *int* OC_level(pOctant octant)

Returns the level of octant in the octree. The root octant is level 0.

OC_numDown int OC_numDown(pOctant)

Returns the number of octants in the tree below the given octant. The number returned does not include the given octant.

OC_isTerminal int OC_isTerminal(pOctant)

Returns 1 if the octant is a terminal octant, 0 otherwise.

OC_bounds void OC_bounds(pOctant octant, double *min[3]*, double *max[3]*)

Returns the coordinates of the two extreme corners of octant. min is the corner that has the minimum x, y and z values, max is the opposite corner.

OC_origin void OC_origin(pOctant octant, double *origin[3]*)

Returns the origin of octant. *origin* is the coordinates of the point in the center of the octant.

OC_numEnt int OC_numEnt(pOctant)

Returns the number of mesh entities in a terminal octant. Returns -1 if the octant is not a terminal octant. The number of entities is the number of regions only in a solid mesh (i.e. the faces, edges, etc. are not included). In a shell mesh it is the number of faces. In a mixed solid/shell mesh it is the number of regions plus the number of "shell" faces.

OC_setNumEnt void OC_setNumEnt(pOctant, int n)

Sets the number of mesh entities that are in a terminal octant. The number of entities is the number of regions only in a solid mesh (i.e. the faces, edges, etc. should not be included). In a shell mesh it would be the number of faces. In a mixed solid/shell mesh it would be the number of regions plus the number of "shell" faces.

OC_setEntity void OC_setEntity(pOctant, pEntity ent, int n)

Sets ent to be the n^{th} mesh entity in the given terminal octant.

OC_entity pEntity OC_entity(pOctant octant, int n)

Returns the n^{th} mesh entity in octant. Returns null if octant is not terminal.

5.2 Octree Operators

OT_new pOctree *OT_new(pMesh mesh)

Creates a new octree object. mesh is the pMesh object that this octree is associated with. The mesh must have already been created and loaded before the octree is loaded using OT_load().

OT_delete void OT_delete(pOctree octree)

Deletes octre, including all of the octants in the octree.

OT_load void OT_load(pOctree, char *filename)

Loads an octree object from the file filename. The filename should be the full file name of the file that contains the octree information (including the .soc extension if there is one).

OT_write

void OT_write(pOctree octree, char *filename, char *progname)

Writes octree to a file called filename (the extension that is used to denote this info is .soc, this extension must be passed as a part of filename). progname is a string containing the name of the program writing the file.

OT_root

pOctant OT_root(pOctree)

Returns the root octant of the octree.

OT_setOrigin

void OT_setOrigin(pOctree, double x, double y, double z)

Sets the origin of the octree to x,y,z.

OT_setRoot

void OT_setRoot(pOctree, pOctant root)

Sets the root octant of the octree to root.

OT_setSize

void OT_setSize(pOctree, double x, double y, double z)

Set the size of the octree (i.e. the size of the root octant) to x,y,z.

6.0 Example Code

This section provides code examples to demonstrate the use of the mesh database.

6.1 Example 1

This example code loops through all the regions in a mesh and prints out the classification of each vertex and node numbers of all the nodes on each region.

```
#include "MSops.h"
#include <stdio.h>

void printRegionInfo(pRegion region);

void main(void)
{
pMesh mesh;
pGModel model;
pRegion region;
int i;
long nr,nf,ne,nv,np;
void *temp=0;

MD_init();                    /* initialize the mesh database */
model = GM_new(0);        /* make a geometric model object */
mesh = MM_new(0,model);   /* make a mesh object */
M_load(model,"test.smd"); /* load the model from the file test.smd */
M_load(mesh,"test.sms");  /* load the mesh from the file test.sms */
MM_makeNodes(mesh,MM_isoparametric);

nr = M_nRegion(mesh); /* get the number of regions, faces, edges, */
nf = M_nFace(mesh);   /* and vertices in the mesh */
ne = M_nEdge(mesh);
nv = M_nVertex(mesh);
np = M_nPoint(mesh);

printf("Number of:\n\tRegions: %ld\n\tFaces: %ld\n\tEdges: %ld\n\tVerticies: %ld\n\tPoints:
        %ld\n\n",nr,nf,ne,nv,np);

for(i=0; i<nr; i++){ /* loop through all the regions in the mesh */
    region = M_nextRegion(mesh,&temp); /* temp=0 to get first region */
     printRegionInfo(region);        /* do something with the region */
```

```
}

M_delete(mesh); /* delete the mesh object (and all mesh entities)*/
M_delete(model);/* delete the model object */

MD_exit();        /* close the mesh database */
}

void printRegionInfo(pRegion region)
{
int j,numface,size;
pFace face;
pPList list;
pVertex vertex;
pNode node;
pEntity gent;

list = R_verticies(region); /* get a list of all the vertices on region */

for(j=0; j < PList_size(list); j++){ /* loop over list */
     vertex = PList_item(list,j);       /* get the j'th vertex */
     gent = V_whatIn(vertex);    /* get the classification of vertex */
     printf("\tVertex %d: whatin: %d, type: %d, ",
               j,MD_tag(gent),EN_type(gent));
     node = EN_node(vertex,0);          /* get node on vertex */
     printf("Node: %d\n",N_num(node));  /* get the node number */
}
printf("\n");
PList_delete(list);      /* delete the list when we're done */
}
```

7.0 Depreciated Operators

These operators should not be used for any new code and old code using them should be updated to use other operators. These operators will be removed in a future version of the mesh database. The operators that shouldbe used instead are given after Use:

R_new

pRegion R_new(int type)

Creates a new pRegion based on type.

Use: **MM_createR**

R_delete

void R_delete(pRegion region)

Deletes region. Does not remove the reference to the region on the faces that bound region. This operator should not be called if the region is part of a mesh (i.e. if it has been attached to a mesh using the M_addRegion operator). In this case the M_removeRegion operator should be called instead so that the mesh knows that the region has been deleted.

Use: **M_removeRegion**

R_setNface

void R_setNface(pRegion region, int n)

Set the number of faces on region. This operator must be called after creating a region before attaching any faces to the region.

Use: **MM_createR**

R_setFace

void R_setFace(pRegion region, pFace face, int dir)

Attach face to region. dir is the direction that region is using the face. dir = 1 means that the face is being used such that it's normal is pointed outside the region. This operator also sets face to point to region.

Use: **MM_createR**

F_new

pFace F_new(eSubType type)

Create a new face of type type.

Use: **MM_createF**

F_delete

void F_delete(pFace face)

Deletes face. Does not remove the reference to the face on the edges that bound the face. This operator should not be called if the face is part of a mesh (i.e. if it has been attached to a mesh using the M_addFace

operator). In this case the M_removeFace operator should be called instead so that the mesh knows that the face has been deleted.

Use: **M_removeFace**

F_setEdge

void F_setEdge(pFace face, pEdge edge, int dir)

Attach edge to face. dir is the direction that face is using edge. dir = 1 means that the edge is being used in the direction that it was defined, from the first vertex to the second vertex of the edge. Edges must be added to the face in the direction of the loop that defines the face.

Use: **MM_createF**

F_setNedge

void F_setNedge(pFace face, int n)

Set the number of edges on face. It is not necessary to call this operator for dynamic faces (it will do nothing in that case). For static faces this operator must be called before attaching edges to the face

Use: **MM_createF**

E_new

pEdge E_new(eSubType type)

Creates a new face of type type.

Use: **MM_createE**

E_delete

void E_delete(pEdge edge)

Deletes edge. Does not remove the reference to the edge on the vertices on the edge. This operator should not be called if the edge is part of a mesh (i.e. if it has been attached to a mesh using the M_addEdge operator). In this case the M_removeEdge operator should be called instead so that the mesh knows that the edge has been deleted.

Use: **M_removeEdge**

E_setVertex

void E_setVertex(pEdge edge, pVertex vertex1, pVertex vertex2)

Sets the vertices on edge to vertex1 and vertex2. The positive direction of the edge is taken to be from vertex1 to vertex2.

Use: **MM_createE**

E_setNface

void E_setNface(pEdge edge, int n)

Sets the number of faces using edge. It is not necessary to call this operator for a dynamic edge (it will do nothing in that situation). For a static

edge this operator must be called before the edge is used in any face definitions.

Use: No longer needed

V_new pVertex V_new(eSubType type)

Create a new vertex of type type.

Use: **MM_createVP**

V_delete void V_delete(pVertex vertex)

Deletes vertex. This operator should not be called if the vertex is part of a mesh (i.e. if it has been attached to a mesh using the M_addVertex operator). In this case the M_removeVertex operator should be called instead so that the mesh knows that the vertex has been deleted.

Use: **M_removeVertex**

V_setNedge void V_setNedge(pVertex vertex, int n)

Set the number of edges using vertex to n. This operator does not need to be called for dynamic vertices, it must be called for static vertices before the vertex is used to define any edges.

Use: no longer needed

M_addRegion void M_addRegion(pModel model, pRegion region)
M_addFace void M_addFace(pModel model, pFace region)
M_addEdge void M_addEdge(pModel model, pEdge region)
M_addVertex void M_addVertex(pModel model, pVertex region)
M_addPoint void M_addPoint(pModel model, pPoint region)

Adds the appropriate entity to the model. This operator just registers that the given entity is a part of the model it does not affect any of the information about that entity.

Use: **MM_createR**, **MM_createF**, **MM_createE**, **MM_createVP** - automatically add the newly created entity to the mesh, no call is needed.