MSTK: Mesh Toolkit, v 1.7 - DRAFT

Rao V. Garimella
T-5, Theoretical Division,
Los Alamos National Laboratory, Los Alamos, NM, USA.
E-mail: rao@lanl.gov

March 1, 2010

1 Introduction

1.1 What is MSTK?

MSTK or Mesh Toolkit is a mesh framework that allows users to represent, manipulate and query unstructured 3D arbitrary topology meshes in a general manner without the need to code their own data structures.

1.2 What MSTK is not?

MSTK is not a mesh generator - it can be used to write more easily than starting from scratch. Also, MSTK cannot answer computational questions related to the mesh (for example, is this point in the given mesh element?).

1.3 Salient Features of MSTK

MSTK is a flexible framework in that it allows a variety of underlying representations for the mesh while maintaining a common interface. It will allow users to choose from different mesh representations either at initialization (implemented) or during the program execution (not implemented) so that the optimal data structures are used for the particular algorithm. The interaction of users and applications with MSTK is through a functional interface that acts as though the mesh always contains vertices, edges, faces and regions and maintains connectivity between all these entities.

MSTK allows for the simultaneous existence of an arbitrary number of meshes. However, any entity in MSTK can belong to only one mesh at a time.

MSTK will eventually support distributed meshes for parallel computing. However, this is still not in place.

To support numerical analysis and other applications, MSTK allows applications to attach application or field data to entities. This data may be integers, reals (doubles), integer vectors, real (double) vectors, integer tensors, real (double tensors) and pointers.

The basis for development of MSTK is laid out in the following paper:

Garimella, R. “Mesh Data Structure Selection for Mesh Generation and FEA Applications,” *International Journal of Numerical Methods in Engineering*, v55 n4, pp. 441-478, 2002.

1.4 Why should I use MSTK?

MSTK offers a flexible infrastructure for representing and manipulating meshes so that mesh-based application developers do not have to deal with the complexitiies of managing their own data structures. The functional interface of MSTK is designed to provide easy access to mesh data and allow for easy-to-read but efficient code to be written for mesh based applications. Accessing the mesh data through the functional interface allows the high level application to be unchanged even if the lower level data structures in MSTK change. MSTK supports a wide array of element types and several representations, all of which can take a new developer years to write and make robust. New users of MSTK can typically start writing their own codes to query and manipulate meshes within a few days of familiarizing themselves with MSTK. Finally, use of MSTK allows easy integration with other codes using MSTK as their mesh data framework.

2 MSTK Concepts

2.1 Unstructured mesh representation

Meshes are made up of topological entities of different dimensions. In a "traditional" 3D finite element mesh, nodes are topological entities called vertices and are topologically 0-dimensional entities, while elements are topological entities called regions and are topologically 3-dimensional entities. In general, meshes can be described using vertices (0-dimensional entities), edges (1-dimensional entities), faces (2-dimensional entities) and regions (3-dimensional entities). MSTK has multiple ways of representing meshes as shown in the below. For example, representation F1 has entities of all dimensions up to the dimension of the mesh while representation R1 has entities of the lowest and highest dimension only. Currently, the application has to choose a particular representation type when a mesh is created although it may choose different representations for different meshes in the same program. In the future, the code will be allowed to switch between different representations depending on the needs of the particular algorithm being executed at that time.

NOTE: CURRENTLY THE F1 REPRESENTATION IS THE MOST ROBUST IMPLEMENTATION SINCE IT IS MOST COMMONLY USED. THIS REPRESENTATION HAS GENERALLY BEEN STABLE FOR SEVERAL YEARS. THE REDUCED REPRESENTATIONS HAVE NOT BEEN AS HEAVILY DEBUGGED AND IT MAY HAVE SOME BUGS. IF YOU CHOOSE TO USE SOME OF THE OTHER REPRESENTATIONS PLEASE REPORT ANY PROB-

LEMS IMMEDIATELY TO `rao@lanl.gov`.

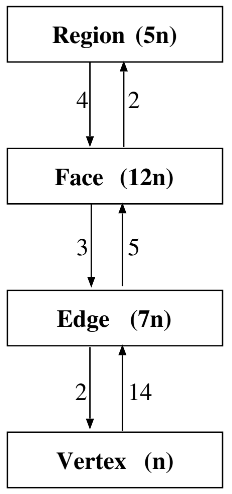
Even when using reduced representations, MSTK can provide the full set of topological adjacencies as if a full representation were being used. When necessary, entities not represented explicitly in the mesh are created on-the-fly and represented as volatile entities. Even though it is possible to take an algorithm that is written for a full representation and use it as is with a reduced representation, application developers are urged to be aware of the costs of each operation for different representations and use this information to design algorithms optimized for each representation.

2.2 Volatile mesh entities in reduced representations

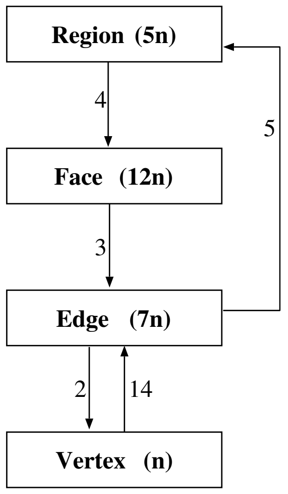
As discussed above reduced representations, do not explicitly represent all types of mesh entities. For example, R1 representations do not explicitly represent edges and faces while R4 representations do explicitly represent edges only. However, whenever an implicit entity is requested from MSTK, the software creates a temporary entity so that it appears that the entity actually exists in the database. Since these entities are created-on-the-fly they are called *volatile* mesh entities. For example, if an application asked for the faces of a region in an R1 representation, MSTK will create as many volatile faces as necessary, put them in a list and return them to the calling application. Thus the application can pretend to work with a full representation (although it is not always efficient to do so).

Volatile entities are stored for a period of time in internal data structures in MSTK. Whenever there is a request for a new volatile mesh entity, the code first checks if this entity has already been created. If the volatile already exists in the database, then the entity is returned as is and if it does not, it is created. This ensures that many copies of the same entity are not created in MSTK. However, to ensure that MSTK also does not store every volatile entity forever (thereby, recreating a full representation), there is a mechanism to perform garbage cleaning on volatile entities. Periodically through the execution of the code, volatile entities that have not been used for a long time are deleted from the database freeing up valuable memory (which was the main point of using a reduced representation after all).

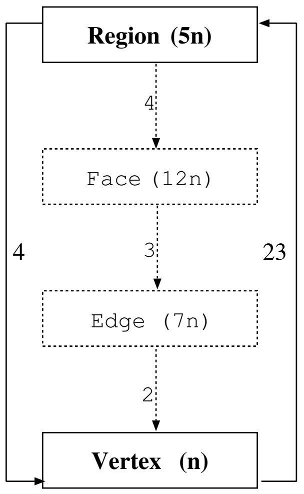
To ensure that garbage cleaning does not delete a volatile mesh entity that is being processed or stored by a calling application, MSTK also provides a mechanism for locking entities. One can specify an autolock mechanism for the entire mesh which means that no volatile entity will ever be deleted once it is created. On the other hand, applications may lock specific entities and unlock them when they cease to be useful.



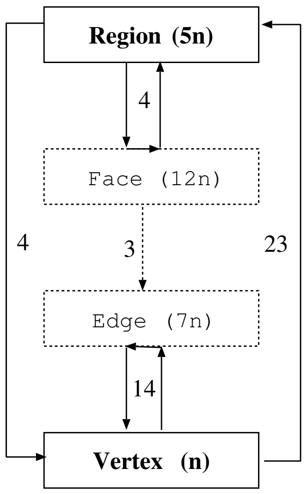
Representation F1



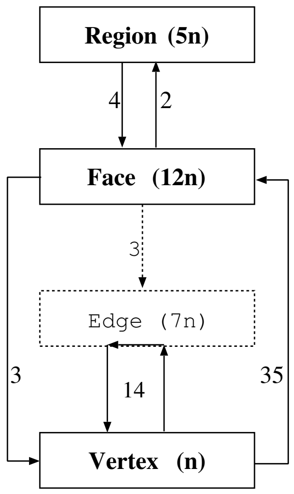
Representation F4



Representation R1



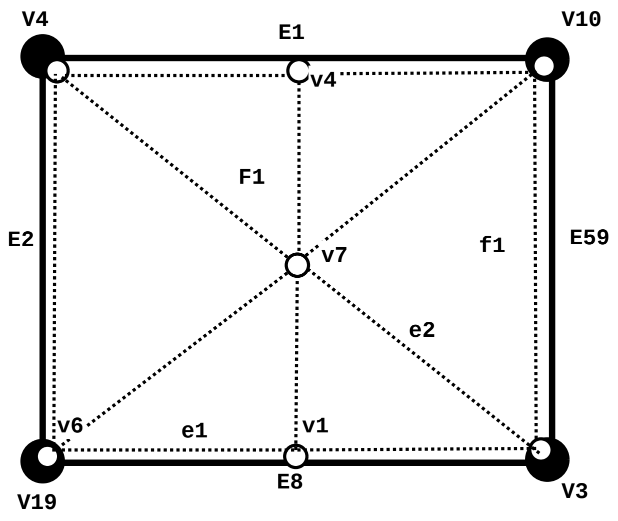
Representation R2



Representation R4

2.3 Mesh entity classification

The concept of mesh entities of various dimensions is actually derived from field of B-rep geometric modeling in which a model is comprised of a hierarchy of topological entities of



```
v6 --> V19 (GEntDim=0, GEntID=19)
v1 --> E8  (GEntDim=1, GEntID=8)
v7 --> F1  (GEntDim=2, GEntID=1)

e1 --> E8  (GEntDim=1, GEntID=8)
e2 --> F1  (GEntDim=2, GEntID=1)

f1 --> F1  (GEntDim=2, GEntID=1)

Lower case letters: mesh entities
Upper case letters: geometric model
                    entities
```

Figure 1: Example mesh and model showing the classification of mesh entities on geometric model entities

different dimensions. *Classification* is the relationship of each mesh entity to the model entity is represents the whole or a part of.

To clarify further, every mesh is a discrete representation of a geometric model. This geometric model may exist in the analyst's mind, as schematic on paper or as a full model in a geometric modeling system. Encoding the relationship of the mesh to this geometric model to the extent possible provides enormous algorithmic benefits to the users of meshes.

Recognizing that one may have varying degrees of information about the geometric model in different situations, MSTK allows mesh entities to store different levels of classification information or no classification information.

Every mesh entity can store the dimension of the model entity it is on (**GEntDim**), the ID of the model entity it is on (**GEntID**) and a pointer to the model entity it is on (**GEntity**). The following picture illustrates the classification of various entities in a simple mesh.

Classification information is very useful to have for simplifying algorithms in meshing and analysis applications. For example, using classification information, one can trivially extract all nodes on the boundary and apply a different smoothing algorithm than nodes in the interior of the domain. An FEA code can apply boundary conditions on nodes and faces more easily if it has classification information.

If the application developer has nothing but the material IDs of the mesh elements, MSTK can build the classification information to the best of its ability using topological and geometric information. However, it must be noted that this is procedure is not perfect and some classification information cannot be derived unambiguously. So, wherever possible, users are encouraged to generate classification information during mesh generation and supply it to MSTK.

2.4 MSTK Application Programming Interface (API)

Even though MSTK is written in C, MSTK is designed in an object oriented manner. Mesh entities, mesh attributes and entity lists are considered to be like objects which have their private data and a set of operators to query and manipulate this data. Even though methods may be devised to circumvent the data hiding in MSTK, since it is written in C, this is highly discouraged. If you think that a particular operator is not efficient or you don'thave access to some mesh data, please contact the MSTK developer for assistance.

An important feature of MSTK is its ability to provide the full set of mesh operators regardless of the representation used. When using reduced representations, entities not represented explicitly in the mesh are returned as volatile objects. However, it is possible to use most MSTK operators on such objects as well.

Currently, MSTK offers only a C API but C++ and Fortran APIs are planned for the near future.

3 MSTK Data Types

List_ptr: Handle to a List object.

Mesh_ptr: Handle to a Mesh object.

MVertex_ptr: Handle to a Mesh Vertex object (Topological Dimension 0).

MEdge_ptr: Handle to a Mesh Edge object (Topological Dimension 1).

MFace_ptr: Handle to a Mesh Face object (Topological Dimension 2).

MRegion_ptr: Handle to Mesh Region object (Topological Dimension 3).

MEntity_ptr: Handle to a generic Mesh Entity object. Any of the above types of entities can be cast as *MEntity_ptr*.

GModel_ptr: Handle to a Geometric Model object.

GEntity_ptr: Handle to a Geometric Entity object.

MAttrib_ptr: Handle to a mesh attribute.

RepType: Enumerated type describing the type of mesh representation. Can be UNKNOWN_REP, F1, F4, R1, R2, R4. See Appendix **??** for schematics of these representations. *Currently only representation types F1 and F4 are supported.*

MType: Enumerated type for mesh entity type. Can be MDELETED, MVERTEX, MEDGE, MFACE, MREGION, MUNKNOWNTYPE, MALLTYPE.

MFType: Enumerated type for mesh face type. Can be FDELETED, FUNKNOWN, TRI, QUAD, POLYGON.

MRType: Enumerated type for mesh region type. Can be RDELETED, RUNKNOWN, TET, PYRAMID, PRISM, HEX, POLYHED.

AttType: Enumerated type for attribute data. Can be INT, DOUBLE or POINTER

4 MSTK Functional Interface

4.1 List

Unordered sets or lists of entities in MSTK are returned as type *List_ptr*. The following are the set operations available in MSTK:

List_ptr **List_New(*int* inisize):** Create a new list with an initial size, *inisize*. If *inisize* is 0, the initial size is set to be 10.

void **List_Delete(*List_ptr* l):** Delete a list.

List_ptr **List_Compress(*List_ptr* l):** Compress a list (delete all the null entries created when entities are removed from the list). Doing this while an algorithm is iterating through the set can currently cause problems!! Calling **List_Compress** could change the pointer for the list due to reallocation.

List_ptr **List_Copy(*List_ptr* l):** Return a copy of a list.

List_ptr **List_Add(*List_ptr* l, *void* *entry):** Add an entry to the list. The entry is strictly appended to the end of the list.

List_ptr **List_ChknAdd(*List_ptr* l, *void* *entry):** Add an entry to a list only if it is not already in the list.

List_ptr **List_Insert(*List_ptr* l, *void* *nuentry, *void* *b4entry):** Insert an entry into the list in the position before 'b4entry'.

List_ptr **List_Inserti(*List_ptr* l, *void* *nuentry, *int* i):** Insert an entry into the list at the i'th valid position and push the entries previously at the i'th and later position back.

int **List_Rem(*List_ptr* l, *void* *entry):** Remove an entry from the list. Returns 1 if successful, 0 otherwise.

int **List_Remi(*List_ptr* l, *int* i):** Remove the i'th valid entry in the list. Returns 1 if successful, 0 otherwise.

int **List_RemSorted(*List_ptr* l, *void* *entry, int (*entry2int)(void *)):** Remove an entry from a list that is sorted according to some correspondence between an entry and an integer value, e.g., **MEnt_ID(entry)** gives the ID of an entity. The mapping between the entry and the integer is given by a call to the routine **entry2int**. The list could be one from which entities have previously been removed. The routine makes use of the sorted nature of the list to locate entries in $O(log(n))$ time where n is the size of the list. An example call to this routine is

fnd = List_RemSorted(mregionslist,region,&(MR_ID));

***int* List_Replace(*List_ptr* l, *void* *entry, *void* *nuentry):** Replace 'entry' with 'nuentry' in set. Returns 1 if successful, 0 otherwise.

***int* List_Replacei(*List_ptr* l, *int* i, void *nuentry):** Replace the i'th valid entry in the list with 'nuentry'. Returns 1 if successful, 0 otherwise.

***int* List_Contains(*List_ptr* l, void *entry):** Returns 1 if list contains the entry, 0 otherwise.

***int* List_Locate(*List_ptr* l, void *entry):** Returns the positional index of the entry in the list. Returns -1 if the list does not contain the entry.

***void* *List_Entry(*List_ptr* l, *int* i):** Return the i'th valid entry in the list. Returns a NULL pointer if the i'th valid entry could not be found.

***void* *List_Next_Entry(*List_ptr* l, *int* *i):** Return the next valid entry in the list. This routine works like an iterator. To start iterating through the list, set the iteration index i=0 and call the routine to get the first entry in the set. Subsequent calls to the routine will iterate through the entries in the list. The routine will return a NULL to indicate that the end of the set is reached.

The value of the iteration index i will be modified by the routine on each call to indicate where in the list it is. This value should not be modified externally while iterating through the list. Also, no specific meaning should be derived from the iteration index by other applications since the internal implementation and interpretation of the index may change at any time.

Finally, there may be unexpected consequences if entries are removed from the list while an iterator is iterating through it using **List_Next_Entry**.

***int* List_Num_Entries(*List_ptr* l):** Return the number of entries in a list.

4.2 Mesh Object

A mesh object is a set of vertices (nodes) possibly connected by other entities such as edges, faces, regions. Depending on the representation chosen and type of mesh, some or all of the entities may be explicitly stored. Full representations contain all types of entities up to the highest dimension of the mesh. For example, a full representation of a tetrahedral mesh contains vertices, edges, faces and regions. However, one type of reduced representation of this mesh may contain only vertices and regions. For a surface mesh, a full representation includes vertices, edges and faces while a reduced representation only has vertices and faces. Also, depending on the type of representation, some adjacencies (information about which entities are connected to which other entities) are stored and others are derived.

***Mesh_ptr* MESH_New(*RepType* type):** Initialize a new mesh object with the given representation type which can be F1, F2, F3, F4, F5, F6, R1, R2, R3, R4. Of these types F1, F4, R1, R2 and R4 are implemented. If the representation type is not known at the present time (e.g. before reading the mesh from a file), the representation type of UNKNOWN_REP can be specified. Note that this only initializes a mesh object, it does not create or generate a mesh which is the work of high level mesh generation routines.

***int* MESH_InitFromFile(*Mesh_ptr* mesh, const char *filename):** Initialize or read a mesh from file into the given mesh object. Returns 1 if successful, 0 otherwise. It is possible to have a an MSTK file in the R1, R2 and R4 format into a mesh initialized as type F1 or F4.

***int* MESH_InitFromGenDesc(*Mesh_ptr* mesh, *int* nv, *double* (*xyz)[3], *int* *nfv, *int* **fvids, *int* nr, *int* *nrv, *int* **rvids, *int* *nrf, *int* ***rfvtemplate):** Initialize a mesh from a minimal description of the mesh passed into the routine. In the routine, **nv** is the number of vertices or nodes and **xyz** is the array of node coordinates. If the mesh is a surface mesh, **nf** is the number of mesh faces, **nfv** gives the number of vertices for each face, **fvids** gives the array indices of the face vertices in ccw manner (starting from 0, not 1). If the mesh is a solid mesh, **nr** specifies the number of solid elements or regions. If the mesh has only standard element types (tets, pyramids, triangular prisms and hexes), then **nrv** indicates the number of vertices of each region and **rvids** gives the array indices of the region vertices. If, on the other hand, the mesh has polyhedral elements, regions have to be described in terms of faces which are in turn described in terms of vertices. Therefore, **nrf** indicates the number of faces for each region and **rfvtemplate** gives the array indices of the vertices for each face of the region.

int **MESH_ImportFromFile(*Mesh_ptr* mesh, const char *filename, const char *format):** Import a mesh data from an external file format and construct MSTK mesh. Currently the only format supported is the GMV[1] file format. The routine imports as many attributes as it can from the input file. It also uses special attributes or keywords as data about element and node classification. For GMV files, the routine uses the "material" data to indicate region or face classification depending on the dimensionality of the mesh. If "material" data is not specified, it uses the "itetclr" attribute to assign region or face classification. It also uses the "icr" keyword describing the number of constraints on a node to interpret if the node is classified on a model region, model face, model edge or a model vertex. The "itetclr" and "icr" keywords are usually present in meshes generated by LAGRIT[2].

int **MESH_BuildClassfn(*Mesh_ptr* mesh):** Build classification information for mesh entities if only partial information is present. In other words, if the only data known is the IDs of geometric model entities ("material regions") of the highest level mesh entities (faces or regions), then this procedure will build information about the type of geometric model entities that the lower dimension entities are on. Faces will be classified as being on a model face (external or interior) or inside model region. Edges will be classified as being on a model edge, model face or in a model region. Vertices will be classified as being on a model vertex, model edge, model face or in a model region. If no classification data is associated with even the highest level entities, the procedure will assume that all the highest level entities are classified on one model entitiy of that dimension. If partial information is available for lower order entities, this routine will not destroy that information. The procedure also tries to detect mesh edges that should be classified on model edges based on the fact that the dihedral angle between the boundary faces connected to the edge is smaller than some tolerance.

int **MESH_DelInterior(*Mesh_ptr* mesh):** Delete the interior of a mesh and retain only its boundaries (including interior boundaries). For solid meshes, this results in a surface mesh (i.e., all mesh regions, and mesh faces, edges and vertices classified on model regions are deleted). For surface meshes, this results in curve mesh. For an edge mesh, only the end vertices are retained (unlikely to be used this way). Undefined for a vertex mesh.

void **MESH_WriteToFile(*Mesh_ptr* mesh, const char *filename):** Save a mesh to a filename. The file is created if it does not exists. It is recommended that the .mstk extension be used for MSTK mesh files. However, there is no such requirement.

[1] `http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html`
[2] `http://lagrit.lanl.gov'`

***void* MESH_ExportToFile(*Mesh_ptr* mesh, const char *filname, const char *format, int natt, char **attnames):** Export a mesh to an external file format. Currently, the only formats supported is the GMV file format (format string: "gmv") and X3D (format string: "flag"). Only integer and double attributes of the mesh are exported to GMV files and no attributes are exported to X3D files.

***void* MESH_ExportToGMV(*Mesh_ptr* mesh, const char *filname, const char *format, int natt, char **attnames):** Export a mesh to a GMV file format.

***void* MESH_ExportToFLAGX3D(*Mesh_ptr* mesh, const char *filname, const char *format, int natt, char **attnames):** Export a mesh to the FLAG X3D format.

***void* MESH_ExportToFLAGX3D_Par(*Mesh_ptr* mesh, const char *filname, const char *format, int natt, char **attnames, int *procids):** Export a mesh to the FLAG X3D format given a partitioned mesh. The owning processor for each element is given in the array **procids**.

***void* MESH_ExportToSTL(*Mesh_ptr* mesh, const char *filname):** Export a mesh to the STL format (SURFACE MESHES ONLY!).

***void* MESH_Tet2Hex(*Mesh_ptr* tetmesh, *Mesh_ptr* *hexmesh):** Convert a tet mesh to a hex mesh by splitting the tets - the quality of the resulting hex mesh is usually not very good.

***void* MESH_Renumber(*Mesh_ptr* mesh):** Renumber the entities of a mesh to avoid gaps in entity IDs. Note that in the current implementation, renumbering mesh entities can make removal of and searching for mesh entities much slower, so this must be avoided as much as possible.

***int* MESH_PartitionWithMetis(*Mesh_ptr* mesh, *int* nparts, *int* **part):** Partition a mesh with the METIS libraries. 'part' is an array that contains the partition number for each mesh face (surface meshes) or mesh region (volume meshes).

NOTE THAT MSTK HAS TO HAVE BEEN COMPILED USING THE COMMAND 'make PAR=1' AND THE CALLING APPLICATION MUST LINK WITH THE METIS LIBRARY.

GModel_ptr **MESH_GModel(*****Mesh_ptr*** **mesh):** Return a handle to the underlying geometric model. If there is no geometric model associated with the mesh, NULL pointer is returned.

RepType **MESH_RepType(*****Mesh_ptr*** **mesh):** Representation type currently being used by the mesh.

char ***MESH_RepType(*****Mesh_ptr*** **mesh):** Representation type currently being used by the mesh returned as a 2-character string.

int **MESH_Num_Vertices(*****Mesh_ptr*** **mesh):** Number of vertices in the mesh.

int **MESH_Num_Edges(*****Mesh_ptr*** **mesh):** Number of edges in the mesh. For reduced representations, this routine returns 0 since it is impractically expensive to count the number of edges when they do not explicitly exist. Applications must find a way to avoid using this routine for reduced representations.

int **MESH_Num_Faces(*****Mesh_ptr*** **mesh):** Number of faces in the mesh. For reduced representations R1 or R2, this routine counts only the faces that are explicitly represented i.e. faces not connected to any mesh region. Therefore, a value of 0 will be returned for the number of faces of a tetrahedral mesh with representation R1 or R2 but the correct number will be reported for a tetrahedral mesh in other representations. Also, the correct number will be reported for the number of faces in a surface mesh in representation R1 or R2. Therefore, this routine must be used carefully.

int **MESH_Num_Regions(*****Mesh_ptr*** **mesh):** Number of regions in the mesh.

MVertex_ptr **MESH_Vertex(*****Mesh_ptr*** **mesh,** ***int*** **i):** Return the i'th vertex in the mesh. Returns NULL if $i < 0$ or $i >$ number of mesh vertices.

MEdge_ptr **MESH_Edge(*****Mesh_ptr*** **mesh,** ***int*** **i):** Return the i'th edge in the mesh. Returns NULL if $i < 0$ or $i >$ number of mesh edges. Returns NULL for reduced representations.

MFace_ptr **MESH_Face(*****Mesh_ptr*** **mesh,** ***int*** **i):** Return the i'th face in the mesh. Returns NULL if $i < 0$ or $i >$ number of mesh faces. Only faces explicitly represented in the mesh are returned for reduced representation (See explanation for MESH_Num_Faces).

MRegion_ptr **MESH_Region(*Mesh_ptr* mesh, *int* i):** Return the i'th region in the mesh. Returns NULL if i < 0 or i > number of mesh region.

MVertex_ptr **MESH_Next_Vertex(*Mesh_ptr* mesh, *int* *idx):** Returns the next vertex while iterating through the vertices of the mesh. See the routine **List_Next_Entry** above for an explanation of how the iteration works. This routine is in general faster than using the routine **MESH_Vertex**.

MEdge_ptr **MESH_Next_Edge(*Mesh_ptr* mesh, *int* *idx):** Returns the next edge while iterating through the edges of the mesh. See the routine List_Next_Entry above for an explanation of how the iteration works. The routine always returns NULL for reduced representations.

MFace_ptr **MESH_Next_Face(*Mesh_ptr* mesh, *int* *idx):** Returns the next face while iterating through the faces of the mesh. See the routine **List_Next_Entry** above for an explanation of how the iteration works. Only faces explicitly represented in the mesh are returned for reduced representation (See explanation for **MESH_Num_Faces**).

MRegion_ptr **MESH_Next_Region(*Mesh_ptr* mesh, *int* *idx):** Returns the next region while iterating through the regions of the mesh. See the routine **List_Next_Entry** above for an explanation of how the iteration works.

MVertex_ptr **MESH_VertexFromID(*Mesh_ptr* mesh, *int* id):** Return mesh vertex with given ID if it exists; return NULL otherwise.

MEdge_ptr **MESH_EdgeFromID(*Mesh_ptr* mesh, *int* id):** Return mesh edge with given ID if it exists; return NULL otherwise. This routine will return NULL for all reduced representations.

MFace_ptr **MESH_FaceFromID(*Mesh_ptr* mesh, *int* i):** Return mesh face with given ID if it exists; return NULL otherwise. If faces are not explicitly represented in the mesh, it will return NULL.

MRegion_ptr **MESH_RegionFromID(*Mesh_ptr* mesh, *int* id):** Return mesh region with given ID if it exists; return NULL otherwise.

int **MESH_Num_Attribs(*Mesh_ptr* mesh):** Number of attributes associated with the mesh.

MAttrib_ptr **MESH_Attrib(*Mesh_ptr* mesh, *int* i):** Return the i'th attribute in the mesh. Returns NULL if i < 0 or i > number of mesh attributes.

MAttrib_ptr **MESH_Next_Attrib(*Mesh_ptr* mesh, *int* *index):** Returns the next attribute while iterating through the attributes of the mesh. See the routine List_Next_Entry above for an explanation of how the iteration works.

MAttrib_ptr **MESH_AttribByName(*Mesh_ptr* mesh, *const char* *name):** Return a mesh attribute with given name if it exisists in the mesh. Returns NULL if mesh has no such attribute.

void **MESH_Set_GModel(*Mesh_ptr* mesh, GModel_ptr geom):** Assign a geometric model handle to the mesh.

void **MESH_Add_Vertex(*Mesh_ptr* mesh, *MVertex_ptr* v):** Add a vertex to the mesh. It is assumed that the vertex and its coordinates set are properly defined. Normally, one need not call this since ***MV_New*** will add the vertex to the mesh. Use this only if you know exactly what you are doing.

void **MESH_Add_Edge(*Mesh_ptr* mesh, *MEdge_ptr* e):** Add an edge to the mesh. It is assumed that the edge is and its topology is defined. Normally, one need not call this since ***ME_New*** will add the edge to the mesh. Use this only if you know exactly what you are doing.

void **MESH_Add_Face(*Mesh_ptr* mesh, *MFace_ptr* f):** Add a face to the mesh. It is assumed that the face and its topology is properly defined. Normally, one need not call this since ***MF_New*** will add the face to the mesh. Use this only if you know exactly what you are doing.

void **MESH_Add_Region(*Mesh_ptr* mesh, *MRegion_ptr* r):** Add a region to the mesh. It is assumed that the region and its topology is properly defined. Normally, one need not call this since ***MR_New*** will add the region to the mesh. Use this only if you know exactly what you are doing.

void **MESH_Rem_Vertex(*Mesh_ptr* mesh, *MVertex_ptr* v):** Remove vertex from mesh. Vertex is not deleted and must be deleted afterward separately. Normally, one need not call this since ***MV_Delete*** will remove the vertex from the mesh. Use this only if you know exactly what you are doing.

void **MESH_Rem_Edge(*Mesh_ptr* mesh, *MEdge_ptr* e):** Remove edge from mesh. Edge is not deleted and must be deleted afterward separately. Normally, one need not call this since ***ME_Delete*** will remove the edge from the mesh. Use this only if you know exactly what you are doing.

void **MESH_Rem_Face(*Mesh_ptr* mesh, *MFace_ptr* f):** Remove face from mesh. Face is not deleted and must be deleted afterward separately. Normally, one need not call this since ***MF_Delete*** will remove the face from the mesh. Use this only if you know exactly what you are doing.

void **MESH_Rem_Region(*Mesh_ptr* mesh, *MRegion_ptr* r):** Remove region from mesh. Region is not deleted and must be deleted afterward separately. Normally, one need not call this since ***MR_Delete*** will remove the region from the mesh. Use this only if you know exactly what you are doing.

void **MESH_Set_AutoLock(*Mesh_ptr* mesh, *int* autolock):** If autolock is 1, all volatile mesh entities created in reduced representations will be automatically locked and cannot be removed by garbage cleaning procedures (They can still be removed by explicitly calling a delete on them). If it 0, volatile mesh entities can be deleted internally after a period of disuse.

int **MESH_AutoLock(*Mesh_ptr* mesh):** Return the autolock status for volatile entities (See **MESH_Set_AutoLock** for details).

4.3 Mesh Vertex Object

***MVertex_ptr* MV_New(*Mesh_ptr* mesh):** Create a new vertex object. No geometric or topological information is embedded in the vertex when it is created. The vertex only knows which mesh it belongs to. The ID of the vertex is set by this function.

***void* MV_Delete(*MVertex_ptr* mvertex, *int* keep):** Delete the vertex. If **keep** is 0, the vertex is removed from the mesh and all topological and geometric information embedded in the vertex is destroyed. If **keep** is 1, the vertex is marked as type *MDELVERTEX* and removed from the mesh but the vertex is not destroyed.

NOTE: *MV_Delete* AND ITS COUNTERPARTS WILL, IN GENERAL, REMOVE AN ENTITY IN $O(log(N))$ TIME WHERE N IS THE TOTAL NUMBER OF ENTITIES ADDED TO THE MESH SINCE THE START OF THE PROGRAM. HOWEVER, IF THE MESH ENTITIES HAVE NOT BEEN RENUUMBERED (EITHER USING MESH_RENUMBER OR EXPLICITLY USING MENT_SET_ID) OR THE MESH ENTITY LISTS HAVE NOT BEEN COMPRESSED THEN THE REMOVAL TIME IS $O(1)$ WHICH IS CLEARLY MUCH SUPERIOR.

***void* MV_Restore(*MVertex_ptr* mvertex):** Restore a deleted vertex. The vertex type is restored from *MDELVERTEX* to *MVERTEX* and the vertex is added back to the mesh.

***void* MV_Set_Coords(*MVertex_ptr* mvertex, double *xyz):** Set the coordinates of the vertex.

***void* MV_Set_GEntity(*MVertex_ptr* mvertex, GEntity_ptr gent):** Set the geometric model entity on which vertex is classified.

***void* MV_Set_GEntDim(*MVertex_ptr* mvertex, *int* gdim):** Set topological dimension of model entity on which vertex is classified.

***void* MV_Set_GEntID(*MVertex_ptr* mvertex, *int* gid):** Set ID of model entity on which vertex is classified.

***void* MV_Add_AdjVertex(*MVertex_ptr* mvertex, *MVertex_ptr* adjvertex):** Add neighboring vertex, adjvertex, to ajdacent vertex list of vertex, mvertex.

***void* MV_Rem_AdjVertex(*MVertex_ptr* mvertex, *MVertex_ptr* adjvertex):** Delete neighboring vertex of given vertex.

***void* MV_Set_ID(*MVertex_ptr* mvertex, *int* id):** Explicitly set ID of a vertex and overwrite the ID set by the MV_New operator. Does not check for duplication of edge IDs.

***Mesh_ptr* MV_Mesh(*MVertex_ptr* mv):** Returns the mesh that this vertex belongs to.

***int* MV_ID(*MVertex_ptr* mvertex):** Returns the ID of the vertex.

***int* MV_GEntDim(*MVertex_ptr* mvertex):** Returns the dimension of the geometric model entity that the vertex is classified on. Returns -1 if not known.

***int* MV_GEntID(*MVertex_ptr* mvertex):** Returns the ID of the geometric model entity that the vertex is classified on. Returns 0 if this information is not known.

***GEntity_ptr* MV_GEntity(*MVertex_ptr* mvertex):** Returns a pointer or handle to the geometric model entity that the vertex is classified on. Returns NULL if this information is not known.

***void* MV_Coords(*MVertex_ptr* mvertex, double *xyz):** Returns the coordinates of the vertex.

***int* MV_Num_AdjVertices(*MVertex_ptr* mvertex):** Returns the number of edge connected neighboring vertices of vertex. *Not efficient for all representations.*

***int* MV_Num_Edges(*MVertex_ptr* mvertex):** Returns the number of edges connected to the vertex.

***int* MV_Num_Faces(*MVertex_ptr* mvertex):** Returns the number of faces connected to the vertex.

***int* MV_Num_Regions(*MVertex_ptr* mvertex):** Returns the number of regions connected to the vertex

***List_ptr* MV_AdjVertices(*MVertex_ptr* mvertex):** List of adjacent or edge connected neighboring vertices of vertex.

List_ptr **MV_Edges(*MVertex_ptr* mvertex):** List of edges connected to the vertex.

List_ptr **MV_Faces(*MVertex_ptr* mvertex):** List of faces connected to the vertex.

List_ptr **MV_Regions(*MVertex_ptr* mvertex):** List of regions connected to the vertex.

4.4 Mesh Edge Object

***MEdge_ptr* ME_New(*Mesh_ptr* mesh):** Create a new edge object. No topological information is embedded in the edge when it is created. The edge only knows which mesh it belongs to. The ID of the edge is set by this function.

***void* ME_Delete(*MEdge_ptr* medge, *int* keep):** Delete the edge. If **keep** is 0, the edge is removed from the mesh and all topological and geometric information embedded in the edge is destroyed. If **keep** is 1, the edge is marked as type *MDELEDGE* and removed from the mesh but the edge is not destroyed. Also, the vertices of this edge no longer point to this edge.

***void* ME_Restore(*MEdge_ptr* medge):** Restore a temporarily deleted edge. The edge type is restored from *MDELEDGE* to *MEDGE* and the edge is added back to the mesh. The vertices of the edge once again point back to the edge.

***void* ME_Set_GEntity(*MEdge_ptr* medge, GEntity_ptr gent):** Set the geometric model entity on which the edge is classified.

***void* ME_Set_GEntDim(*MEdge_ptr* medge, *int* gdim):** Set the topological dimension of model entity on which edge is classified.

***void* ME_Set_GEntID(*MEdge_ptr* medge, *int* gid):** Set ID of model entity on which edge is classified.

***void* ME_Set_GInfo_Auto(*MEdge_ptr* medge):** Derive the classification (GEntDim, GEntID) of the mesh edge automatically (if possible) from the classification of its vertices. If it is not possible to unambiguously get this information, the procedure will keep the default information. **THIS MAY GET WRONG OR INCORRECT INFORMATION WHEN THERE IS AMBIGUOUS DATA. FOR EXAMPLE, IF THE VERTICES OF A MESH EDGE ARE CLASSIFIED ON TWO DIFFERENT MODEL VERTICES, IT IS NOT POSSIBLE TO KNOW WHAT ENTITY THE MESH EDGE IS CLASSIFIED USING JUST THE VERTEX INFORMATION.**

***void* ME_Set_ID(*MEdge_ptr* medge, *int* id):** Explicitly set ID of an edge and overwrite the ID set by the ME_New function. Does not check for duplication of edge IDs.

void **ME_Set_Vertex(*MEdge_ptr* medge, *int* i, *MVertex_ptr* vertex):** Set the i'th vertex of the edge. i can be 0 or 1.

void **ME_Replace_Vertex(*MEdge_ptr* medge, *MVertex_ptr* vert, *MVertex_ptr* nuvert):** Replace i'th vertex by new vertex.

Mesh_ptr **ME_Mesh(*MEdge_ptr* medge):** Returns the mesh that this edge belongs to.

int **ME_ID(*MEdge_ptr* medge):** Returns the ID of the vertex. Returns -1 if not known.

int **ME_GEntDim(*MEdge_ptr* medge):** Returns the dimension of the geometric model entity that the vertex is classified on. Returns -1 if not known.

int **ME_GEntID(*MEdge_ptr* medge):** Returns the ID of the geometric model entity that the vertex is classified on. Returns 0 if this information is not known.

GEntity_ptr **ME_GEntity(*MEdge_ptr* medge):** Returns a pointer or handle to the geometric model entity that the vertex is classified on. Returns NULL if this information is not known.

int **ME_Num_Faces(*MEdge_ptr* medge):** Returns the number of faces connected to the edge.

int **ME_Num_Regions(*MEdge_ptr* medge):** Returns the number of regions connected to the edge.

MVertex_ptr **ME_Vertex(*MEdge_ptr* medge, *int* i):** Returns the i'th vertex of the edge. i=0 returns the first vertex and i=1 returns the second vertex.

MVertex_ptr **ME_OppVertex(*MEdge_ptr* medge, *MVertex_ptr* ov):** Return the vertex opposite to given vertex in edge.

int **ME_UsesEntity(*MEdge_ptr* medge, *MEntity_ptr* mentity, *int* etype):** Check if edge uses given lower dimension entity, *mentity.* The dimension of the entity is specified by the *etype* variable. For an edge, the only lower dimensional entity is a vertex. If the edge uses the vertex, the function returns 1; otherwise it returns 0. If any other type of entity is specified, the function returns 0.

List_ptr **ME_Faces(*MEdge_ptr* medge):** Returns the set of faces using this edge.

List_ptr **ME_Regions(*MEdge_ptr* medge):** Returns the set of regions using this edge.

MEdge_ptr **MVs_CommonEdge(*MVertex_ptr* v1, *MVertex_ptr* v2):** Return the edge connecting vertices v1 and v2, if it exists. If such an edge does not exist, the function returns 0.

double **ME_Len(*MEdge_ptr* e):** Return the length of the straight line connecting the two vertices of the edge.

double **ME_LenSqr(*MEdge_ptr* e):** Return the square of the length of the straight line connecting the two vertices of the edge.

void **ME_Vec(*MEdge_ptr* e, double *evec):** Return the vector going from the first vertex of the edge to the second vertex of the edge.

int **MEs_AreSame(*MEdge_ptr* e1, *MEdge_ptr* e2):** *Applicable only to reduced representations.* Check if two edges described by only their vertices are the same. In a reduced representation, edges are temporary objects described by their end vertices. If two temporary edges are described by the same vertices, the edges are considered to be the same.

void **ME_Lock(*MEdge_ptr* e):** Lock a volatile edge so that it cannot be deleted by garbage cleaning procedures (It can still be deleted by **ME_Delete**.

void **ME_UnLock(*MEdge_ptr* e):** Unlock a volatile edge so that it may be deleted freely by garbage cleaning procedures.

4.5 Mesh Face Object

***MFace_ptr* MF_New(*Mesh_ptr* mesh):** Create a new face object. No topological information is embedded in the face when it is created. The face only knows which mesh it belongs to. The ID of the face is set by this function.

***void* MF_Delete(*MFace_ptr* mface):** Delete the face. If **keep** is 0, the face is removed from the mesh and all topological and geometric information embedded in the face is destroyed. If **keep** is 1, the face is marked as type *MDELFACE* and removed from the mesh but the face is not destroyed. Also, the vertices/edges of this face no longer point up to this face.

***void* MF_Restore(*MFace_ptr* mface):** Restore a temporarily deleted face. The face type is restored from *MDELFACE* to *MFACE* and the face is added back to the mesh. The vertices/edges of the face once again point back to the face.

***void* MF_Set_GEntity(*MFace_ptr* mface, GEntity_ptr gent):** Set the geometric model entity on which the face is classified.

***void* MF_Set_GEntDim(*MFace_ptr* mface, *int* gdim):** Set the dimension of the geometric model entity on which the face is classified.

***void* MF_Set_GEntID(*MFace_ptr* mface, *int* gid):** Set the ID of the geometric model entity on which the face is classified.

***void* MF_Set_GInfo_Auto(*MFace_ptr* mface):** Derive the classification (GEntDim, GEntID) of the mesh face automatically (if possible) from the classification of its vertices or edges. If it is not possible to unambiguously get this information, the procedure will keep the default information. **THIS MAY GET WRONG OR INCORRECT INFORMATION WHEN THERE IS AMBIGUOUS DATA.**

***void* MF_Set_ID(*MFace_ptr* mface, *int* id):** Explicitly set ID of a face and overwrite the ID set by the **MF_New** operator. Does not check for duplication of face IDs.

***void* MF_Set_Edges(*MFace_ptr* mface, *int* n, *MEdge_ptr* *edges, *int* *dirs):** Set the edges of the face along with their directions. The ordered set of edge pointers and their directions are passed in through arrays along with the number of edges. The edges are assumed to be ordered clockwise around the face. If an edge direction is along the clockwise direction of the face then the entry in the 'dirs' array must be 1; otherwise it must be 0. This function is relevant only for full representations in MSTK.

void **MF_Set_Vertices(*MFace_ptr* mface, *int* n, *MVertex_ptr* *verts):** Set the vertices of the face. The ordered set of vertices (ccw around the face) is passed in through an array along with the number of vertices. This routine will collect/build all lower order topological information (edges, edge directions in the face) as needed by the face.

void **MF_Replace_Edges(*MFace_ptr* mface, *int* nold, *MEdge_ptr* *oldedges, *int* nnu, *MEdge_ptr* *nuedges):** Replace a set of edges in the face with another set of edges. The direction in which the new edges are to be used in the face are automatically deduced. This function is relevant only for full representations in MSTK. The old and new edge sets must be comprised of edges connected to each other and the end vertices of the new set must match the end vertices of the new set.

void **MF_Replace_Edges_i(*MFace_ptr* mface, *int* nold, *int* i, *int* nnu, *MEdge_ptr* *nuedges):** Replace the *nold* edges in the face starting with the i'th edge and going ccw around the face with a new set of edges. The directions in which the new edges are used in the face are automatically deduced. This function is relevant only for full representations in MSTK.

void **MF_Replace_Vertex(*MFace_ptr* mface, *MVertex_ptr* mvertex, *MVertex_ptr* nuvertex):** Replace a vertex in the face with another vertex. This function is relevant only for reduced representations in MSTK.

void **MF_Replace_Vertex_i(*MFace_ptr* mface, *int* i, *MVertex_ptr* nuvertex):** Replace the i'th vertex in the face with a new vertex. This function is relevant only for reduced representations in MSTK.

void **MF_Insert_Vertex(*MFace_ptr* mface, *MVertex_ptr* nuvertex, *MVertex_ptr* b4vertex):** Insert a vertex in the face before **b4vertex** w.r.t. a ccw ordering of the vertex faces.

void **MF_Insert_Vertex_i(*MFace_ptr* mface, *MVertex_ptr* nuvertex, *int* i):** Insert a new vertex before vertex i in the face w.r.t. a ccw ordering of vertices.

Mesh_ptr **MF_Mesh(*MFace_ptr* mf):** Returns the mesh that this mesh belongs to.

int **MF_ID(*MFace_ptr* mface):** Returns the ID of the face. Returns 0 if not known.

int **MF_GEntDim(*MFace_ptr* mface):** Returns the dimensions of the geometric model entity that the vertex is classified on. Returns -1 if not known.

int **MF_GEntID(*MFace_ptr* mface):** Returns the ID of the geometric model entity that the vertex is classified on. Returns 0 if this information is not known.

GEntity_ptr **MF_GEntity(*MFace_ptr* mface):** Returns a pointer or handle to the geometric model entity that the vertex is classified on. Returns NULL if this information is not known.

int **MF_Num_Vertices(*MFace_ptr* mface):** Returns the number of vertices of the face.

int **MF_Num_Edges(*MFace_ptr* mface):** Returns the number of edges of the face.

int **MF_Num_AdjFaces(*MFace_ptr* mface):** Returns the number of adjacent faces of a face. This operator is relevant only in planar or surface meshes, i.e., for boundary faces not connected to any regions.

List_ptr **MF_Vertices(*MFace_ptr* mface, *int* dir, *MVertex_ptr* mvert):** Return the ordered set of the vertices of the face. The vertices are ordered in ccw direction while looking down the face 'normal', if 'dir' is 1 and in the cw direction, if 'dir' is 0. If 'mvert' is specified, the vertex set is reordered so that it is the first vertex (This argument will be added soon to the function. For now, omit this argument). The behavior of this function can be illustrated using Figure 2. For the face shown in the figure, a vertex set with ccw ordering or 'dir' = 1 is V_0, V_1, V_2, V_3 and a vertex set with cw ordering or 'dir' = 0 is V_0, V_3, V_2, V_1. A vertex set with ccw ordering starting with vertex V_2 is V_2, V_3, V_0, V_1.

List_ptr **MF_Edges(*MFace_ptr* mface, *int* dir, *MVertex_ptr* mvert):** Return the ordered set of edges of the face. The edges are ordered in the ccw while looking down the face 'normal' if dir is 1 and in the cw if dir is 0. If 'mvert' is specified, the edge set is reordered so that the first edge in the set contains this vertex. More precisely, if 'dir' is 1, and the first edge is 'e' used in the face in direction 'd', then **ME_Vertex(e,!d)** = mvert. With reference to Figure 2, the edges of the face in the ccw direction or 'dir' = 1 are E_0, E_1, E_2, E_3 and in the opposite dir are E_3, E_2, E_1, E_0. If 'mvert' or the starting vertex is specified as V_2, the edge set in ccw direction or 'dir' = 1 is E_2, E_3, E_0, E_1 and in the opposite direction is E_1, E_0, E_3, E_2.

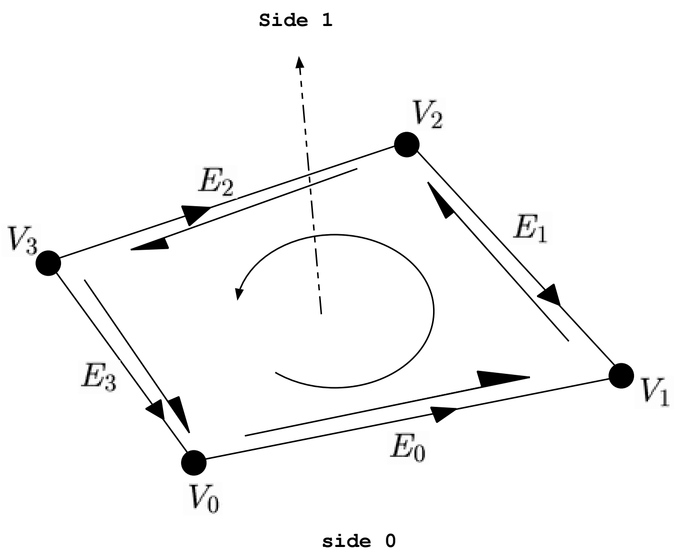


Figure 2: Face definition

List_ptr **MF_AdjFaces(*MFace_ptr* mface):** List of adacent faces of a face. This operator is relevant only in planar or surface surface meshes, i.e., for boundary faces not connected to any regions.

int **MF_EdgeDir(*MFace_ptr* mface, *MEdge_ptr* medge):** Returns the direction in which the face uses the given edge. If the faces use the edge in the positive direction, the function returns 1; otherwise it returns 0.

int **MF_EdgeDir_i(*MFace_ptr* mface, *int* i):** Returns the direction in which the face uses its i'th edge. If the face uses the edge in the positive direction the function returns 1; otherwise it returns 0;

List_ptr **MF_Regions(*MFace_ptr* mface):** Return the set of regions connected to the face. If the face is not used by any regions, the function returns NULL to indicate that the set is empty. If not the set may contain one or two regions.

MRegion_ptr **MF_Region(*MFace_ptr* mface, *int* side):** Returns the region on the specified side of the face. The positive side of the face (side = 1) is the side towards which the face normal points. The negative side of the face (side = 0) is the opposite side.

int **MF_UsesEntity(*MFace_ptr* mface, *MEntity_ptr* mentity, *int* type):** Check if the face uses the given lower dimension entity, 'mentity'. The type of the entity is specified by the 'etype' variable. For a face, a lower dimensional entity can be a vertex or an edge. If the face uses the vertex or the edge, the function returns 1; otherwise it returns 0. If any other type of entity is specified, the function returns 0.

MFace_ptr **MVs_CommonFace(*int* nv, *MVertex_ptr* *fverts):** Return a face connected to all the given mesh vertices. Returns NULL if no such face exists.

MFace_ptr **MEs_CommonFace(*int* ne, *MEdge_ptr* *fedges):** Return a face connected to all the given mesh edges. Returns NULL if no such face exists.

int **MFs_AreSame(*MFace_ptr* f1, *MFace_ptr* f2):** *Applicable only to reduced representations.* Check if two faces described by vertices are equivalent. For example, a face described by vertices $\{1, 2, 3\}$ is equivalent to faces described by vertices $\{1, 3, 2\}$ and $\{3, 1, 2\}$.

void **MF_Coords(*MFace_ptr* mface, *int* *n, double (*xyz)[3], *int* dir):** Returns the coordinates of the face vertices in an array along with the number of vertices.

void **MF_Lock(*MFace_ptr* f):** Lock a volatile face so that it cannot be deleted by garbage cleaning procedures (It can still be deleted by **MF_Delete**.

void **MF_UnLock(*MFace_ptr* f):** Unlock a volatile face so that it may be deleted freely by garbage cleaning procedures.

4.6 Mesh Region Object

***MRegion_ptr* MR_New(*Mesh_ptr* mesh):** Create a new region object. No topological information is embedded in the region when it is created. The region only knows which mesh it belongs to. The ID of the region is set by this function.

***void* MR_Delete(*MRegion_ptr* mregion, *int* keep):** Delete the region. Deletes all topological information embedded in the region. If **keep** is 0, the region is removed from the mesh and all topological and geometric information embedded in the region is destroyed. If **keep** is 1, the region is marked as type *MDELREGION* and removed from the mesh but the region is not destroyed. Also, the vertices, edges and faces of this region no longer point up to this region.

***void* MR_Restore(*MRegion_ptr* mregion):** Restore a temporarily deleted region. The region type is restored from *MDELREGION* to *MREGION* and the region is added back to the mesh. The vertices, edges and faces of the region once again point back to the region.

***void* MR_Set_GEntity(*MRegion_ptr* mregion, GEntity_ptr gent):** Set the geometric model entity on which the region is classified.

***void* MR_Set_GEntDim(*MRegion_ptr* mregion, *int* gdim):** Set the dimension of the geometric model entity on which the region is classified.

***void* MR_Set_GEntID(*MRegion_ptr* mregion, *int* gid):** Set the ID of the geometric model entity on which the region is classified.

***void* MR_Set_GInfo_Auto(*MRegion_ptr* mregion):** Derive the classification (GEntDim, GEntID) of the mesh region automatically (if possible) from the classification of its vertices or faces. If it is not possible to unambiguously get this information, the procedure will keep the default information. **THIS MAY GET WRONG OR INCORRECT INFORMATION WHEN THERE IS AMBIGUOUS DATA.**

***void* MR_Set_ID(*MRegion_ptr* mregion, *int* id):** Explicitly set ID of a region and overwrite the ID set by the **MR_New** function. Does not check for duplication of region IDs.

***void* MR_Set_Faces(*MRegion_ptr* mregion, *int* nf, *MFace_ptr* *mfaces, *int* *dirs):** Set the faces of the region along with their directions. The *unordered* set of faces and their directions are passed in through arrays along with the number of faces. If the normal of the face points out of the region, the associated direction to be passed in is 1; otherwise it is 0. This function is only relevant for full representations in MSTK.

void **MR_Set_Vertices(*MRegion_ptr* mregion, *int* nv, *MVertex_ptr* *mvertices, *int* nf, *int* **rfv_template):** Set the vertices of the region. This routine will collect/build all lower order topological information (edges, edge directions in the face) as needed by the face. For standard elements, the vertices must be ordered as indicated in Appendix C, while **nf** can be 0 and **rfv_template** can be NULL. For non-standard elements, **nf** is the number of faces of the polyhedron and **rfv_template** gives the vertices used in each face. More precisely, **rfv_template** has **nfv** pointers to integer array i. The first entry of each **rfv_template**$[i]$ represents the number of vertices in that face and the remaining entries represent the list of vertices of the face listed so that the face defined by them points out of the region.

void **MR_Replace_Face(*MRegion_ptr* mregion, *MFace_ptr* mface, *MFace_ptr* nuface, *int* dir):** Replace a face of the region with another face. The direction in which the new face is used in the region must also be supplied. This function is only relevant for full representations in MSTK.

void **MR_Replace_Vertex(*MRegion_ptr* mregion, *MVertex_ptr* mvertex, *MVertex_ptr* nuvertex):** Replace a vertex of a region with another vertex. This function is relevant only for reduced representations in MSTK.

void **MR_Replace_Face_i(*MRegion_ptr* mregion, *int* i, *MFace_ptr* mface, *int* dir):** Replace the i'th face in the region with another face.The direction in which the new face is used in the region must also be supplied. This function is only relevant for full representations in MSTK.

void **MR_Replace_Vertex_i(*MRegion_ptr* mregion, *int* i, *MVertex_ptr* mvertex):** Replace the i'th vertex of the region with another vertex. This function is only relevant for reduced representations in MSTK.

Mesh_ptr **MR_Mesh(*MRegion_ptr* mregion):** Returns the mesh that the region belongs to.

int **MR_ID(*MRegion_ptr* mregion):** Returns the ID of the region. Returns 0 if not known.

int **MR_GEntDim(*MRegion_ptr* mregion):** Returns the dimension of the geometric model entity the region is classified on. Always returns 3 since a mesh region can be classified only on a model region.

int **MR_GEntID(*MRegion_ptr* mregion):** Returns the ID of the geometric model entity that the region is classified on. Returns 0 if not known.

GEntity_ptr **MR_GEntity(*MRegion_ptr* mregion):** Returns a pointer or handle to the geometric model entity that the vertex is classified on. Returns NULL if this information is not known.

int **MR_Num_Vertices(*MRegion_ptr* mregion):** Returns the number of vertices of a region.

int **MR_Num_Edges(*MRegion_ptr* mregion):** Returns the number of edges of a region.

int **MR_Num_Faces(*MRegion_ptr* mregion):** Returns the number of faces of a region.

int **MR_Num_AdjRegions(*MRegion_ptr* mregion):** Returns the number of adjacent regions of a region, i.e., regions sharing a face with this region.

List_ptr **MR_Vertices(*MRegion_ptr* mregion):** Returns the set of vertices of a region. For standard elements the vertices are ordered as indicated in Appendix C. For non-standard elements the set is unordered.

List_ptr **MR_Edges(*MRegion_ptr* mregion):** Return the unordered set of edges of a region.

List_ptr **MR_Faces(*MRegion_ptr* mregion):** Returns the set of faces of a region.

List_ptr **MR_AdjRegions(*MRegion_ptr* mregion):** Returns the set of adjacent regions of a region, i.e., regions sharing a face with this region. The set is not ordered.

int **MR_FaceDir(*MRegion_ptr* mregion, *MFace_ptr* mface):** Returns the direction in which the region uses the given face. Returns 1 if the face normal points out of the region and returns 0 if the face normal points into the region.

***int* MR_FaceDir_i(*MRegion_ptr* mregion, *int* i):** Returns the direction in which the region uses the i'th face. Returns 1 if the face normal points out of the region and returns 0 if the face normal points into the region.

***int* MR_UsesEntity(*MRegion_ptr* mregion, *MEntity_ptr* ment, *int* type):** Check if the region uses the given lower dimension entity, 'mentity'. The type of the entity is

***void* MR_Coords(*MRegion_ptr* mregion, *int* *n, double (*xyz)[3]):** Returns the coordinates of the region vertices in an array along with the number of vertices. For standard elements, the ordering is as given in Appendix C. For non-standard elements, the ordering is arbitrary.

4.7 Generic Entity Object

The following functions operate on generic mesh entities of type *MEntity_ptr*. This implies that variables of type *MVertex_ptr*, *MEdge_ptr*, *MFace_ptr*, *MRegion_ptr* can all be passed in place of *MEntity_ptr* variables in the following functions.

***int* MEnt_ID(*MEntity_ptr* mentity):** Returns the ID of a generic entity.

***int* MEnt_Dim(*MEntity_ptr* mentity):** Returns the topological dimension or type of generic entity.

***int* MEnt_OrigDim(*MEntity_ptr* mentity):** Returns the original topological dimension or type of generic entity before it was temporarily deleted.

***int* MEnt_IsVolatile(*MEntity_ptr*) mentity):** Is the entity a temporary (volatile) entity in a reduced representation or an explicitly represented persistent entity. For example, this query will return 1 on any edge in a reduced representation.

***Mesh_ptr* MEnt_Mesh(*MEntity_ptr* mentity):** Returns the mesh that the entity belongs to.

***int* MEnt_GEntDim(*MEntity_ptr* mentity):** Returns the dimension of the geometric model entity that the entity is classified on.

***GEntity_ptr* MEnt_GEntity(*MEntity_ptr* mentity):** Returns a pointer or handle to geometric model entity that the entity is classified on.

***void* MEnt_Delete(*MEntity_ptr* mentity, *int* keep):** Delete a generic mesh entity. If **keep** is 0, the entity is removed from the mesh and all topological and geometric information embedded in the entity is destroyed. If **keep** is 1, the region is marked as deleted and removed from the mesh but the entity is not destroyed. Also, lower order entities in the mesh no longer point up to this entity.

***void* MEnt_Set_AttVal(*MEntity_ptr* ent, *MAttrib_ptr* attrib, *int* ival, *double* lval, *void* *pval):** Set the value of an attribute for given mesh entity. Depending on whether the attribute is integer, double or pointer, the appropriate entry in the argument list is specified.

void **MEnt_Rem_AttVal(*MEntity_ptr* ent, *MAttrib_ptr mattrib*:** Clear the value of the given attribute from the entity.

int **MEnt_Get_AttVal(*MEntity_ptr* ent, *MAttrib_ptr* int *ival, double *rval, double **pval):** Get the value of an attribute from an entity. Depending on the attribute type, either **ival**, **rval** or **pval** is returned.

void **MEnt_Rem_AllAttVals(*MEntity_ptr*):** Remove all attribute values attached to an entity.

4.8 Mesh Attributes

MAttrib_ptr **MAttrib_New(*Mesh_ptr* mesh, *const char* *att_name, *MAttType* att_type, *MType* entdim):** Define a new mesh attribute. Along with the mesh to which this attribute is assigned (**mesh**) and the name of the attribute (**att_name**), the type of the attribute (**att_type**) must be specified as **INT**, **DOUBLE** or **POINTER**. Also, the dimension of the entity for which the attribute value can be set must be specified as **MVERTEX**, **MEDGE**, **MFACE**, **MREGION** or **MALLTYPE**.

char ***MAttrib_Get_Name(*MAttrib_ptr* attrib, *char* *att_name):** Get the name of the given attribute.

MAttType **MAttrib_Get_Type(*MAttrib_ptr* attrib):** Get the type of the attribute (Can return **INT**, **DOUBLE**, **PVAL**).

MType **MAttrib_Get_EntDim(*MAttrib_ptr* attrib):** Get the dimension (or type) of entity attribute can be asssigned to. Can be **MVERTEX**, **MEDGE**, **MFACE**, **MREGION** or **MALLTYPE**.

void **MAttrib_Delete(*MAttrib_ptr* attrib):** Delete an attribute.

4.9 Entity Marks

Entity marks or markers are a way of tagging entities. Such functionality is useful in algorithms which must keep track of processed entities to avoid duplication of work. An example of such an operation is creating a union of entity sets while extracting upward adjacency information such as the regions connected to an edge. Use of entity marks avoids calling **List_ChknAdd** (search through the list and add item if it is not there) or using attributes to tag entities with 0 or 1. As a result it is much more efficient than either option.

***int* MSTK_GetMarker():** Returns a unique marker ID which may be used to tag entities.

***void* MEnt_Mark(*MEntity_ptr* ent, *int* mkr):** Mark an entity with the given marker 'mkr'.

***int* MEnt_IsMarked(*MEntity_ptr* ent, *int* mkr):** Check if an entity is marked with the given marker 'mkr'.

***void* MEnt_Unmark(*MEntity_ptr* ent, *int* mkr):** Unmark an entity with respect to the given marker 'mkr'

***void* List_Mark(*List_ptr* list, *int* mkr):** Mark a set of entities with given marker.

***void* List_Unmark(*List_ptr* list, *int* mkr):** Unmark a set of entities with respect to the given marker.

***void* MSTK_FreeMarker(*int* mkr):** Release the marker ID given by **MSTK_GetMarker()** so that it can be reused. Care must be taken to unmark all entities marked with this marker ID before releasing it. If not, subsequent operations with reassigned marker will find a tag on some entities and mistake them for being processed.

An example use of entity marks is given below:

```
/* Code to get the faces of a vertex */
/* NOTE: This code is for illustration of entity marks only. */
/*       This functionality is already available through MV_Faces */

vedges = MV_Edges(v);      /* Edges of vertex */

mkid = MSTK_GetMarker();   /* Get a new marker */
vfaces = List_New(10);     /* Initialize list of faces cncted to v */

idx = 0;
while ((ve = List_Next_Entry(vedges,&idx))) {   /* For each edge */

  efaces = ME_Faces(ve);                /* Get faces of edge */
  idx1 = 0;
  while ((ef = List_Next_Entry(efaces,&idx1))) {
    if (!MEnt_IsMarked(ef,mkid)) {         /* Is the face already in list? */
      List_Add(vfaces,ef);                 /* No? Add it to the list and   */
      MEnt_Mark(ef,mkid);                  /* mark it as being in the list */
    }
  }
  List_Delete(efaces);

}

List_Delete(vedges);

List_Unmark(vfaces,mkid);                 /* Unmark all the marked faces */
MSTK_FreeMarker(mkid);                    /* Very important to free the marker!! */
                                          /* otherwise, we'll run out of markers */
```

4.10 Mesh Modification

int **ME_Swap2D(*MEdge_ptr* e, *MEdge_ptr* *enew, *MFace_ptr* fnew[2]):** Swap an edge in a triangular mesh. No checks are performed for topological or geometric validity.

MVertex_ptr **MVs_Merge(*MVertex_ptr* v1, *MVertex_ptr* v2):** Merge two vertices, v1 and v2, and return the retained vertex. By default, v1 is retained.

MFace_ptr **MFs_Join(*MFace_ptr* f1, *MFace_ptr* f2, *MEdge_ptr* e):** Join two faces along common edge and create new face by eliminating the common edge as shown in Figure 3. If 'f1' has 'n1' edges and 'f2' has 'n2' edges, then the new face has ('n1'+'n2'-2) edges.

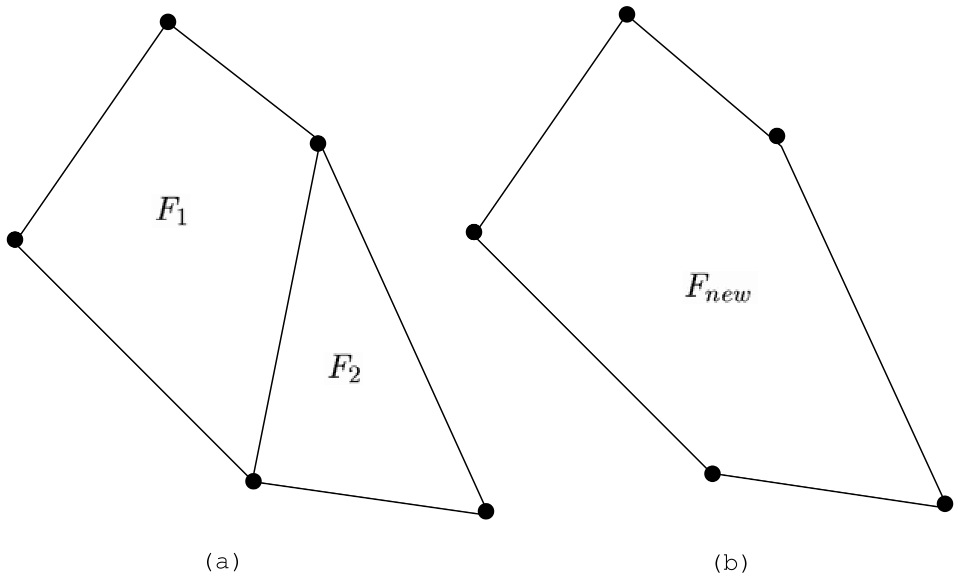


Figure 3: Joining two faces (a) Two faces F_1 and F_2 sharing a common edge (b) New pentagonal face F_{new} created by eliminating the common edge.

4.11 Utilities

***void* MSTK_Report(*char* *module, *char* *message, *ErrType* severity):** Error handler for MSTK. 'module' is the name of the function in which the error occurs. 'message' is the error message and is recommended to be less than 1024 characters in length. 'severity' is an error code and can be MESSG, WARN, ERROR or FATAL. If the error code is FATAL, the program will quit after printing the error. If the same message is repeated successively, then the message is printed only the first time.

***void* List_PrintID(*List_ptr* l):** Debugging utility to print the IDs of the entities in a set.

***void* MV_Print(*MVertex_ptr* v, int lev):** Debugging utility to print information about a mesh vertex, **v**. The argument **lev** controls the level of detail of the information printed. **lev** = 0 prints the minimum information, i.e., vertex pointer, its ID and its coordinates. If **lev** = 1, the function prints classification information for the vertex (if available), i.e., ID and dimension of the model entity that the vertex is on. If **lev** > 1, then upward detailed adjacency information is also printed for the vertex, i.e., information is printed about the edges, faces and regions connected to the vertex.

***void* ME_Print(*MEdge_ptr* e, int lev):** Debugging utility to print information about a mesh edge, **e**. The argument **lev** controls the level of detail of the information printed. **lev** = 0 prints the minimum information, i.e., edge pointer, its ID and the IDs of its two vertices. If **lev** = 1, the function prints classification information for the edge (if available), i.e., ID and dimension of the model entity that the edge is on. Also, more detailed vertex information printed in this case. If **lev** > 1, the function prints detailed upward adjacency information for the edge, i.e., information is printed about the faces and regions connected to the edge.

***void* MF_Print(*MFace_ptr* f, int lev):** Debugging utility to print information about a mesh face, **f**. The argument **lev** controls the level of detail of the information printed. **lev** = 0 prints the minimum information, i.e., the face pointer and its ID. If **lev** = 1, the function prints classification information for the edge (if available), i.e., ID and dimension of the model entity that the face is on. Also, a signed list of the edges of the face is printed. If **lev** > 1, the function prints detailed downward and upward adjacency information for the face, i.e., information is printed about the edges and vertices of the face, and about the regions connected to the face.

***void* MR_Print(*MRegion_ptr* r, int lev):** Debugging utility to print information about a mesh region, **r**. The argument **lev** controls the level of detail of the information printed. **lev** = 0 prints the minimum information, i.e., region pointer and its ID. If **lev** = 1, the function prints classification information for the region (if available), i.e., ID of the model entity that the region is on. Also, a signed list of the faces of the region is printed. If **lev** > 1, the function prints detailed downward adjacency information for the region, i.e., information is printed about the faces, edges and vertices forming the region.

A Getting meshes in and out of MSTK

There are a few ways of getting mesh data into MSTK. The first is to prepare an MSTK file with the full connectivity of an F1 representation and to read it in using MESH_InitFromFile. This is typically a difficult task for many applications which only have element-node data. In such cases, one can prepare an MSTK file in the R1 file format (in which the node listing is followed by the region-node connectivity) and read it in to any of the other representations, say F1. The other possibility is to write a file in the GMV format and use MESH_ImportFromFile to import the mesh in.

MSTK can write meshes out to different formats including GMV, STL and FLAG X3D. When compiled with the appropriate options and using a partitioner from an external library, MSTK can also write out a parallel X3D file.

B Using MSTK in a numerical simulation code

In addition to mesh generation and mesh manipulation algorithms, MSTK can be used to write mesh-based numerical simulation codes such as Finite element, finite volume and finite difference methods. MSTK is particularly suited to developing methods which operate on general unstructured meshes. There is some overhead to using operators to access mesh data instead of directly accessing arrays; however, we believe that the ease of managing complex mesh data through an API outweighs this cost. Without a framework like MSTK, applications have to devise their own unstructured mesh data structures and the code is littered with direct accesses to these data structures (often complicated array-based data structures). This makes even small changes to the data structure very onerous because thousands to tens of thousands of lines of code are affected. In the end, individual developers create adhoc patches (often duplicated in different ways) to overcome any deficiencies in the data structures and eventually the code becomes unmanageable. We believe this is poor software engineering.

B.1 Simulation Variables

When using MSTK in a physics based code, the question of storing physics variables (node velocities, element pressures) is an important one.

There are two ways of storing physics in combination with MSTK. The first is to use the attributes feature in MSTK. Scalar variables can be stored with entities as double precision values of mesh attributes; vector and tensor variables can be stored as pointer variables. When storing vectors as pointers, it is more efficient to allocate a large array to store the vectors for all entities and store pointers into that large array with the individual entities. This way the application does not allocate a huge number of small arrays. This method is very convenient as the variables are readily available using the handle of the entity, i.e., by calling **MEnt_Get_AttVal**. However, this method could be somewhat inefficient and also requires a paradigm shift in how codes are written.

The other way of managing physics variables is to directly allocate linear arrays for each variable and using the entity ID to reference entries in this array. For example, one might have a linear array **rho** for the densities of all elements and we use **rho[MR_ID(r)-1]** to find the density of a particular element **r**. The ID of elements is obtained quite easily and stays static unless **MESH_Renumber** is called explicitly. If there is mesh modification, there will be intermediate parts of this array which will no longer be relevant but this is easily managed. This method is closer to traditional approaches and can be adopted quite easily. It could be more efficient than using attributes but is less "object-oriented."

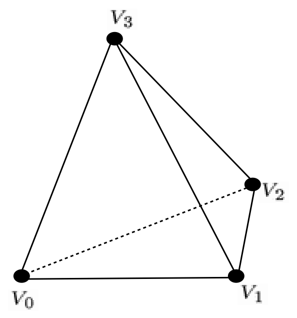
B.2 Boundary Conditions, Material Properties

Applying boundary conditions in an MSTK-based simulation code can be much easier than in a "traditional" simulation code, *if* the mesh carries the right classification information. In a "traditional" simulation code, boundary conditions are specified using one of two ways:

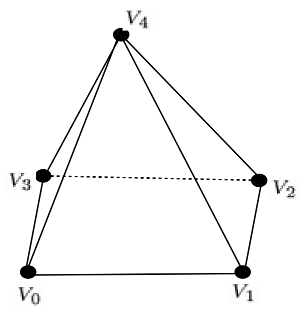
- An implicit geometric entity is defined and the boundary condition is specified on that geometry, e.g., normal velocity is 1.0 for all nodes on the plane [1.35 2.7695 1]. This requires one to check the coordinates of all nodes to see if they lie on this plane and apply the boundary condition on those nodes.
- Entity sets like node sets or face sets are defined as a collection of entity IDs on which a particular boundary conditions are applied. The user has pre-selected a set of nodes and applied the boundary condition one them.

MSTK, on the other hand, provides for an easier way to specify boundary conditions, initial conditions and material properties provided the mesh has classification information. For example, if we knew that model face 51 has a pressure boundary condition on it, we only need to find all faces that are classified on model face 51 (MF_GEntDim(f)=2, MF_GEntID(f)=51) and specify the pressure on that face in the code. The input file for the boundary conditions does not have to explicitly list any face sets nor do we have to use error prone geometric checks on the entire mesh to apply boundary conditions. In fact, if the mesh is directly tied to a geometric model (using a GEntity_ptr), it is possible to specify a boundary condition on a complex surface such as a Trimmed NURBS surface and still apply the condition without error-prone geometric checks or large input files. Similarly, material properties can be specified per geometric model region and applied automatically to every mesh region classified in that model region without the need for element sets in input files.

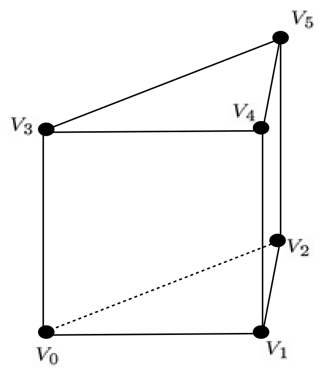
C Conventions for Vertex, Edge Numbering in Standard Region Types



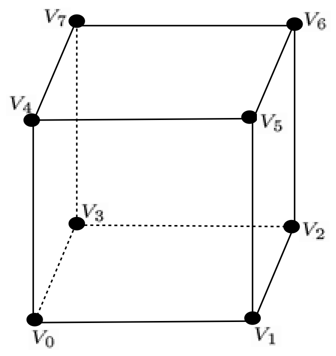
(a)
Tetrahedron



(b)
Pyramid



(c)
Triangular Prism



(d)
Hexahedron

D MSTK File Format

D.1 MSTK ASCII File Format

# This is a comment
# The string "MSTK" and File version number (1.0)

MSTK Ver

*# char *reptype - Type of representation*
int *NV, NE, NF, NR - Number of vertices, edges, face, regions*

RepType NV NE NF NR

```
# VERTEX INFO
# Each record has
# double X,Y,Z Coordinates
# int Mdim - Topological type or dimension of model entity that
#            the vertex is on
# int Mid - ID of model entity the vertex is on
# Mdim and Mid can be -1 and 0 resp. if model info. is absent

X_Coord  Y_Coord  Z_Coord  Mdim  Mid
X_Coord  Y_Coord  Z_Coord  Mdim  Mid
. . .
# Repeated NV times

# EDGE INFO - present only if NE ≠ 0
# Keyword 'edges' followed by edge records # Each edge record has
# int Vid_1, Vid_2 - IDs of first, second vertex of edge
# int Mdim, Mid

edges Vid_1  Vid_2  Mdim  Mid
Vid_1  Vid_2  Mdim  Mid
. . .
# Repeated NEdges times
```

# **FACE INFO** - *present only if NF $\neq$ 0*
# *Keyword 'faces' # char *FLtype: Keyword for lower order entity describing faces*
# *Values: Vertex, Edge (case insensitive), e.g. VeRteX or EDGE*

faces FLtype

# *If face described by vertices, then each face record has*
# int *NFV - Number of face vertices*
# int *Vid_1 - ID of first vertex of face*
# int *Vid_2 - ID of second vertex of face*
. . .
# int *Vid_1 - ID of NFV'th vertex of face*
# int *Mdim, Mid*

```
NFV   Vid_1   Vid_2   ...   Vid_NFV   Mdim   Mid
NFV   Vid_1   Vid_2   ...   Vid_NFV   Mdim   Mid
. . .
```
# *Repeated NFaces times*

# *If face described by edges, then each face record has*
# int *NFE - Number of face edges*
# int ±*Eid_1 - signed ID of first edge of face*
# int ±*Eid_2 - signed ID of second edge of face*
. . .
# int ±*Eid_NFE - signed ID of NFE'th edge of face*
# int *Mdim, Mid*
#
# *if sign of edge is +, face uses edge in direction it was defined*
# *if sign of edge is -, face uses edge in opposite direction*

```
NFE   ±Eid_1   ±Eid_2   ...   ±Eid_NFE   Mdim Mid
NFE   ±Eid_1   ±Eid_2   ...   ±Eid_NFE   Mdim Mid
. . .
```
# *Repeated NFaces times*

# **REGION INFO** - *present only if NR $\neq$ 0*
# *Keyword 'regions' # char *RLtype - keyword for lower order entity describing region*

```
# Values: Vertex, Face (case insensitive), e.g. VERtex or faCE

regions RLtype

# if region described by vertices, then each region record has
# int NRV - Number of region vertices
# int Vid_1 - ID of first vertex of region
# int Vid_2 - ID of second vertex of region
# . . .
# int Vid_NFE - ID of NRV'th vertex of region
# int Mid, (NOTE: Mdim is not specified, since it has to be 3)

NRV   Vid_1   Vid_2   ...   Vid_NRV   Mid
NRV   Vid_1   Vid_2   ...   Vid_NRV   Mid
. . .
# Repeat NR times


# if region described by faces, then each region record has
# int NRF - Number of region faces
# int Fid_1 - signed ID of first face of region
# int Fid_2 - signed ID of second face of region
# . . .
# int Fid_NRF - signed ID of NRF'th face of region
# int Mdim, Mid
#
# if sign of face is +, face normal points out of region
# if sign of edge is -, face normal points into region

NRF   ±Fid_1   ±Fid_2   ...   ±Fid_NRF   Mid
NRF   ±Fid_1   ±Fid_2   ...   ±Fid_NRF   Mid
. . .
# Repeated NR times

# NOT IMPLEMENTED
# VERTEX ATTRIBUTES
# int NVA - Number of Vertex attributes
#
# char *VA_name_1 - Name of first vertex attribute
# int VA_type_1 - Type of first vertex attribute
# int VA_dim_1 - Dimension of first vertex attribute
#
# char *VA_name_2 - Name of second vertex attribute
# int VA_type_2 - Type of second vertex attribute
# int VA_dim_2 - Dimension of first vertex attribute
#
# . . .
#
# char *VA_name_NVA - Name of NVA'th vertex attribute
# int VA_type_NVA - Type of NVA'th vertex attribute
# int VA_dim_NVA - Dimension of NVA'th vertex attribute
#
# VA_type can be 1 (int), 2 (double), 3 (string)
# VA_dim = 1 for scalars, VA_dim = length of vector for vector
# VA_dim can only be 1 when VA_type is string

NVA
VA_name_1   VA_type_1   VA_dim_1
VA_name_2   VA_type_2   VA_dim_2
.
VA_dim_NVA   VA_type_NVA   VA_dim_2

# For each vertex attribute record, set of attribute values
# E.G., there are 3 attributes for each vertex:
# a scalar int, a vector of 3 doubles and a string

VA_int   VA_double_1   VA_double_2   VA_double_3   VA_string
VA_int   VA_double_1   VA_double_2   VA_double_3   VA_string
.
.
.
# Repeated NV times
# EDGE ATTRIBUTES
# Similar to Vertex attribute description

NEA
EA_name_1   EA_type_1   EA_dim_1
EA_name_2   EA_type_2   EA_dim_2
.
EA_dim_NEA   EA_type_NEA   EA_dim_2

# For each edge attribute record, set of attribute values
# E.G., a scalar int, a scalar double and a string

EA_int   EA_double   EA_string
EA_int   EA_double   EA_string
. . .
# Repeated NE times

# FACE ATTRIBUTES
# Similar to Vertex attribute description

NFA
FA_name_1   FA_type_1   FA_dim_1
FA_name_2   FA_type_2   FA_dim_2
.
FA_dim_NFA   FA_type_NEA   FA_dim_2

# For each face attribute record, set of attribute values
# E.G., a vector of 2 doubles and a string

FA_double_1   FA_double_2
FA_double_1   FA_double_2
. . .
# Repeated NF times
# REGION ATTRIBUTES
# Similar to Vertex attribute description

NRA
RA_name_1   RA_type_1   RA_dim_1
RA_name_2   RA_type_2   RA_dim_2
.
RA_dim_NRA   RA_type_NRA   RA_dim_NRA

# For each Region attribute record, set of attribute values
# E.G., a vector of 3 ints

RA_int_1   RA_int_2   RA_int_3
RA_int_1   RA_int_2   RA_int_3
. . .
# Repeated NR times
```

E Example program

NOTE: This program is included in the distribution.

```
#include <stdio.h>
#include <stdlib.h>
#include "MSTK.h"
#include "test.h"

int main(int argc, char *argv[]) {
 int i, idx, idx2, ok, edir, nv, ne;
 double xyz[3];
 char meshname[256];
 Mesh_ptr mesh;
 MVertex_ptr v;
 MEdge_ptr e;
 MFace_ptr f;
 GEntity_ptr gent;
 List_ptr fedges;


 /* Initialize MSTK - Always do this even if it does
    not seem to matter in this version of MSTK */

 MSTK_Init();

 /* Load the mesh */

 strcpy(meshname,argv[1]);
 strcat(meshname,".mstk");

 mesh = MESH_New(UNKNOWN_REP);
 ok = MESH_InitFromFile(mesh,meshname);
 if (!ok) {
   fprintf(stderr,"Cannot file input file %s\n\n\n",meshname);
   exit(-1);
 }
```

```
/* Print some info about the mesh */

nv = MESH_Num_Vertices(mesh);
for (i = 0; i < nv; i++) {
 v = MESH_Vertex(mesh,i);

 /* Basic info */
 printf("\n");
 printf("Vertex: 0x%-x   ID: %-d   ",v,MV_ID(v));

 /* Classification w.r.t. geometric model */

 if (MV_GEntDim(v) == -1)
   fprintf(stderr,"Unknown Classification\n");
 else {
  printf("GEntID: %-d   GEntDim: %-d\n",MV_GEntID(v),MV_GEntDim(v));
  if ((gent = MV_GEntity(v)))
    printf("Model entity pointer: 0x%-x\n",gent);
 }

 /* Coordinates */
 MV_Coords(v,xyz);
 printf("Coords: %16.8lf %16.8lf %16.8lf\n",xyz[0],xyz[1],xyz[2]);
}

idx = 0;
while (f = MESH_Next_Face(mesh,&idx)) {

 /* Basic info */
 printf("\n");
 printf("Face: 0x%-x   ID: %-d   ",f,MF_ID(f));

 /* Classification w.r.t. geometric model */

 if (MF_GEntDim(f) == -1)
  fprintf(stderr,"Unknown Classification\n");
 else {
  printf("GEntID: %-d   GEntDim: %-d\n",MF_GEntID(f),MF_GEntDim(f));
```

```
    if ((gent = MF_GEntity(f)))
     printf("Model entity pointer: 0x%-x\n",gent);
   }
   printf("\n");

   /* Edges of face */
   fedges = MF_Edges(f,1,0);
   ne = List_Num_Entries(fedges);
   printf("Edges: %-d\n",ne);
   printf("Object       ID     GEntID  GEntDim   Vertex IDs\n");
   idx2 = 0; i = 0;
   while (e = List_Next_Entry(fedges,&idx2)) {
    edir = MF_EdgeDir_i(f,i);
    if (edir)
     printf("0x%-8x    %-8d %-8d     %-1d       %-d  %-d\n",
            e,ME_ID(e),ME_GEntID(e),ME_GEntDim(e),
            MV_ID(ME_Vertex(e,0)),MV_ID(ME_Vertex(e,1)));
    else
     printf("0x%-8x    %-8d %-8d     %-1d       %-d  %-d\n",
            e,-ME_ID(e),ME_GEntID(e),ME_GEntDim(e),
            MV_ID(ME_Vertex(e,0)),MV_ID(ME_Vertex(e,1)));
    i++;
   }
   printf("\n");
   List_Delete(fedges);
  }

  /* Write out a copy of the mesh */

  strcpy(meshname,argv[1]);
  strcat(meshname,"-copy.mstk");
  MESH_WriteToFile(mesh,meshname,F1);

  /* No need to delete a mesh if program ends right afterwards */

  MESH_Delete(mesh);
  return 1;
 }
```